

```
1|
2| Enabling Source Code
3|
4| ---- NOTICE ----
5|
6| All source code is
7| Copyright (c) 1995-2002
8| Columbia Data Products, Inc.
9| All Rights Reserved.
10|
11|
12|
13| PSM COM Object Provider Source
14|
15| The COM object provider allows a web browser of other
16| COM enabled object to access the COM server located on
   | the same or another machine. --LPW
17|
18|
19|
20| File Listing: CDP.h
21|
22| #define VER_COMPANYNAME_STR    "Columbia Data
   | Products, Inc."
23| #define VER_LEGALTRADEMARKS_STR "PSM\256 is a
   | trademark of " VER_COMPANYNAME_STR
24|
25| #define VER_LEGALCOPYRIGHT_YEARS "1995-2001"
26| #define VER_LEGALCOPYRIGHT_STR  "Copyright \251 "
   | VER_LEGALCOPYRIGHT_YEARS " " VER_COMPANYNAME_STR
27|
28| #if DBG
29| #define VER_DEBUG                VS_FF_DEBUG
30| #else
31| #define VER_DEBUG                0
32| #endif
33|
34| #if BETA
35| #define VER_PRODUCTBETA_STR      "BETA"
36| #define VER_PRERELEASE           VS_FF_PRERELEASE
37| #else
38| #define VER_PRERELEASE           0
39| #define VER_PRODUCTBETA_STR      ""
40| #endif
41|
42| #define VER_FILEFLAGSMASK        VS_FFI_FILEFLAGSMASK
43| #define VER_FILEOS               VOS_NT_WINDOWS32
44| #define VER_FILEFLAGS            (VER_PRERELEASE |
```

```

| VER_DEBUG)
45|
46|
47|
48| File Listing: psmprov.c
49|
50| #include <stdio.h>
51| #include <stdlib.h>
52| #include <string.h>
53| #include <stddef.h>
54| #include <windows.h>
55| #include <tchar.h>
56| #include <process.h>
57| #include <time.h>
58| #include <direct.h>
59| #include <lm.h>
60|
61| #include <winioctl.h>
62| #include "..\include\psmoem.h"
63|
64| __declspec( dllexport ) ULONG __cdecl Validate(void)
65| {
66|     return 0x11223344;
67| }
68|
69| __declspec( dllexport ) BOOLEAN __cdecl IsProvVendor(IN
| WCHAR *wVendorName)
70| {
71|     if (_wcsicmp(wVendorName,_PsmVendorNameWStr)==0)
| return TRUE;
72|     return FALSE;
73| }
74|
75| BOOL WINAPI DIIMain(
76|     HINSTANCE hDllInst,
77|     DWORD fdwReason,
78|     LPVOID lpvReserved)
79| {
80| #if 0
81|     // Dispatch this call based on the reason it was
| called.
82|     switch (fdwReason) {
83|         case DLL_PROCESS_ATTACH:
84|         case DLL_THREAD_ATTACH:
85|         case DLL_PROCESS_DETACH:
86|         case DLL_THREAD_DETACH:
87|     }
88| #endif
89|
90|     return TRUE;

```



```

91| }
92|
93|
94|
95| File Listing: RESOURCE.h
96|
97| //{NO_DEPENDENCIES}
98| // Microsoft Developer Studio generated include file.
99| // Used by SBPSMAN.RC
100| //
101| #define VER_PRODUCTBUILD          3
102| #define VER_PRODUCTVERSION_W      0x0101
103|
104| // Next default values for new objects
105| //
106| #ifdef APSTUDIO_INVOKED
107| #ifndef APSTUDIO_READONLY_SYMBOLS
108| #define _APS_NO_MFC                1
109| #define _APS_NEXT_RESOURCE_VALUE    101
110| #define _APS_NEXT_COMMAND_VALUE     40001
111| #define _APS_NEXT_CONTROL_VALUE     1000
112| #define _APS_NEXT_SYMED_VALUE      101
113| #endif
114| #endif
115|
116|
117|
118| PSM COM Object Server Source
119|
120| The COM server allows a remote COM provider to access
    | PSM functionality over a network. --LPW
121|
122|
123|
124| File Listing: cluster.c
125|
126| #include <windows.h>
127| #include <stdio.h>
128| #include <stdlib.h>
129| #include <process.h>
130| #include <tchar.h>
131| #include <winioctl.h>
132| #include <undoc.h>
133| #include "psm.h"
134|
135| #include "..\driver\ioctl.h"
136| #include <clusapi.h>
137| #include "cluster.h"
138|
139| #define DLOG(x) printf x

```

```

140|
141| ULONG ClusterGetApis( pClusApis Apis )
142| {
143|     ULONG Err=0;
144|
145|     memset(Apis,0,sizeof(tClusApis));
146|
147|     Apis->ClusApi=LoadLibrary(TEXT("clusapi.dll"));
148|
149|     if((Apis->ClusApi!=INVALID_HANDLE_VALUE) &&
        | (Apis->ClusApi!=NULL)) {
150|         Apis->OpenCluster=
        | (tOpenCluster)GetProcAddress(Apis->ClusApi,"OpenCluster"
        | );
151|         Apis->CloseCluster=
        | (tCloseCluster)GetProcAddress(Apis->ClusApi,"CloseCluste
        | r");
152|         Apis->GetClusterKey=
        | (tGetClusterKey)GetProcAddress(Apis->ClusApi,"GetCluster
        | Key");
153|         Apis->ClusterRegCreateKey =
        | (tClusterRegCreateKey)GetProcAddress(Apis->ClusApi,"Clus
        | terRegCreateKey");
154|         Apis->ClusterRegOpenKey =
        | (tClusterRegOpenKey)GetProcAddress(Apis->ClusApi,"Cluste
        | rRegOpenKey");
155|         Apis->ClusterRegDeleteKey =
        | (tClusterRegDeleteKey)GetProcAddress(Apis->ClusApi,"Clus
        | terRegDeleteKey");
156|         Apis->ClusterRegCloseKey =
        | (tClusterRegCloseKey)GetProcAddress(Apis->ClusApi,"Clust
        | erRegCloseKey");
157|         Apis->ClusterRegEnumKey =
        | (tClusterRegEnumKey)GetProcAddress(Apis->ClusApi,"Cluste
        | rRegEnumKey");
158|         Apis->ClusterRegSetValue =
        | (tClusterRegSetValue)GetProcAddress(Apis->ClusApi,"Clust
        | erRegSetValue");
159|         Apis->ClusterRegDeleteValue =
        | (tClusterRegDeleteValue)GetProcAddress(Apis->ClusApi,"Cl
        | usterRegDeleteValue");
160|         Apis->ClusterRegQueryValue =
        | (tClusterRegQueryValue)GetProcAddress(Apis->ClusApi,"Clu
        | sterRegQueryValue");
161|         Apis->ClusterRegEnumValue =
        | (tClusterRegEnumValue)GetProcAddress(Apis->ClusApi,"Clus
        | terRegEnumValue");
162|         Apis->ClusterRegQueryInfoKey =
        | (tClusterRegQueryInfoKey)GetProcAddress(Apis->ClusApi,"C
        | lusterRegQueryInfoKey");

```

```

163|     Apis->ClusterRegGetKeySecurity =
| (tClusterRegGetKeySecurity)GetProcAddress(Apis->ClusApi,
| "ClusterRegGetKeySecurity");
164|     Apis->ClusterRegSetKeySecurity =
| (tClusterRegSetKeySecurity)GetProcAddress(Apis->ClusApi,
| "ClusterRegSetKeySecurity");
165|     Apis->GetNodeClusterState =
| (tGetNodeClusterState)GetProcAddress(Apis->ClusApi,"GetN
| odeClusterState");
166|
167| #ifdef DEBUG
168|     // make sure everything loaded
169|     {
170|         ULONG i;
171|         ULONG *Ptr;
172|         Ptr = (ULONG*)Apis;
173|         for(i=0;i<sizeof(tClusApis)/4;i++,Ptr++) {
174|             if(!Ptr) {
175|                 DLOG((TEXT("Error! Api %d is
| missing!\n"),i));
176|             }
177|         }
178|     }
179| #endif
180| } else {
181|     Err = GetLastError();
182|     DLOG((TEXT("Error %08x load module\n"),Err));
183| }
184| return Err;
185| }
186|
187| ULONG ClusterCopyKey( pClusApis Apis, HKEY LocalKey,
| HKEY ClusterKey )
188| {
189|     ULONG Err=0;
190|     WCHAR Name[100];
191|     ULONG NameLen;
192|     ULONG Index=0;
193|     ULONG Type;
194|     ULONG DataSize;
195|     PVOID Data;
196|
197|     do {
198|         NameLen = sizeof(Name) / sizeof(WCHAR);
199|         DataSize=0;
200|         Err =
| RegEnumValueW(LocalKey,Index,Name,&NameLen,NULL,NULL,NUL
| L,&DataSize);
201|         if(!Err) {
202|             Data = LocalAlloc(LPTR,DataSize);

```

```

203|         if(Data) {
204|             Err =
| RegQueryValueExW(LocalKey,Name,NULL,&Type,Data,&DataSize
| );
205|             if(!Err) {
206|                 Err =
| Apis->ClusterRegSetValue(ClusterKey,Name,Type,Data,DataS
| ize);
207|                 if(!Err) {
208|                     DLOG((TEXT("Success updating
| %d:'%S', size=%d cluster\n"),Type,Name,DataSize));
209|                 } else {
210|                     DLOG((TEXT("Error %08x setting
| data for value %S\n"),Err,Name));
211|                 }
212|             } else {
213|                 DLOG((TEXT("Error %08x getting data
| for value %S\n"),Err,Name));
214|             }
215|             LocalFree(Data);
216|         } else {
217|             DLOG((TEXT("Error! out of memory for
| data, need %d bytes\n"),DataSize));
218|             Err = ERROR_OUTOFMEMORY;
219|         }
220|     } else {
221|         if(Err!=ERROR_NO_MORE_ITEMS) {
222|             DLOG((TEXT("Error %08x enumerating
| key\n"),Err));
223|         }
224|     }
225|     Index++;
226| } while(Err==0);
227|
228| if(Err==ERROR_NO_MORE_ITEMS) {
229|     Err = 0;
230| }
231| return Err;
232| }
233|
234| ULONG CopyKeyToClusterKey( WCHAR *VolumeGuid, WCHAR
| *Uniqueld)
235| {
236|     tClusApis Apis;
237|     HCLUSTER Cluster;
238|     ULONG Err=0;
239|     HKEY ClusterKey;
240|     HKEY PSMKey;
241|     HKEY psmlocalkey;
242|     HKEY volumelocalkey;

```

```

243|  ULONG Disp=0;
244|
245|  DLOG((TEXT("VolumeGuid='%S',
    | uniqueid='%S'\n"),VolumeGuid,Uniqueld));
246|
247|  Err = ClusterGetApis(&Apis);
248|  if(!Err) {
249|      Cluster = Apis.OpenCluster(NULL);
250|      if(!Cluster) {
251|          DLOG((TEXT("unable to open cluster\n")));
252|          Err = GetLastError();
253|      }
254|      if(!Err) {
255|          ClusterKey =
    | Apis.GetClusterKey(Cluster,KEY_ALL_ACCESS);
256|          if(!ClusterKey) {
257|              DLOG((TEXT("unable to open root
    | key\n")));
258|              Err = GetLastError();
259|          }
260|          if(!Err) {
261|              WCHAR Temp[200];
262|
    | wcscpy(Temp,L"PersistentStorageManager\\");
263|              wscat(Temp,Uniqueld);
264|
265|              __try {
266|                  Err = Apis.ClusterRegCreateKey(
267|                      ClusterKey,
268|                      Temp,
269|                      REG_OPTION_NON_VOLATILE,
270|                      KEY_ALL_ACCESS,
271|                      NULL,
272|                      &PSMKey,
273|                      &Disp
274|                  );
275|              } __except (EXCEPTION_EXECUTE_HANDLER)
    | {
276|                  Err = GetExceptionCode();
277|                  DLOG((TEXT("Exception %08x in
    | ClusterRegCreateKey\n"),Err));
278|              }
279|
280|              if(!Err) {
281|                  DLOG((TEXT("Disposition of PSM key
    | is %08x\n"),Disp));
282|                  // delete old keys, incase they
    | have been deleted locally
283|                  if(Disp==REG_OPENED_EXISTING_KEY )
    | {

```

```

284|             if((Err =
| Apis.ClusterRegDeleteValue(PSMKey,L"HeaderMap")) {
285|                 DLOG((TEXT("Error %08x
| deleting header map\n"),Err));
286|             }
287|             if((Err =
| Apis.ClusterRegDeleteValue(PSMKey,L"IndexMap")) {
288|                 DLOG((TEXT("Error %08x
| deleting index map\n"),Err));
289|             }
290|             if((Err =
| Apis.ClusterRegDeleteValue(PSMKey,L"CacheMap")) {
291|                 DLOG((TEXT("Error %08x
| deleting cache map\n"),Err));
292|             }
293|         }
294|
295|         Err =
| RegOpenKeyExW(HKEY_LOCAL_MACHINE,
296| | L"SYSTEM\\CurrentControlSet\\Services\\PSMan5",
297|         0,
298|         KEY_READ,
299|         &psmlocalkey);
300|         if(!Err) {
301|             Err =
| RegOpenKeyExW(psmlocalkey,
302| | VolumeGuid,
303|         0,
304|         KEY_READ,
305|         &volumelocalkey);
306|             if(!Err) {
307|                 Err =
| ClusterCopyKey(&Apis,volumelocalkey,PSMKey);
308|
| RegCloseKey(volumelocalkey);
309|             } else {
310|                 DLOG((TEXT("Cluster: Error
| %08x opening volume guid key\n"),Err));
311|             }
312|             RegCloseKey(psmlocalkey);
313|         } else {
314|             DLOG((TEXT("Cluster: Error %08x
| opening psman5 key\n"),Err));
315|         }
316|         Apis.ClusterRegCloseKey(PSMKey);
317|     } else {
318|         DLOG((TEXT("Cluster: Error %08x
| creating PSM key\n"),Err));
319|     }

```

```

320|         Apis.ClusterRegCloseKey(ClusterKey);
321|     } else {
322|         DLOG((TEXT("Cluster: Error %08x getting
    | cluster key\n"),Err));
323|     }
324|     Apis.CloseCluster(Cluster);
325| } else {
326|     DLOG((TEXT("Cluster: Error %08x opening
    | cluster\n"),Err));
327| }
328|     FreeLibrary(Apis.ClusApi);
329| } else {
330|     DLOG((TEXT("Cluster: Error %08x unable to get
    | entry points\n"),Err));
331| }
332|     return Err;
333| }
334|
335|
336| ULONG DoCleanClusterDatabaseEvent( WCHAR *VolumeGuid,
    | WCHAR *Uniqueld )
337| {
338|     return CopyKeyToClusterKey(VolumeGuid, Uniqueld);
339| }
340|
341|
342|
343| File Listing: cluster.h
344|
345| typedef LONG
346| (WINAPI *tClusterRegCreateKey)(
347|     IN HKEY hKey,
348|     IN LPCWSTR lpszSubKey,
349|     IN DWORD dwOptions,
350|     IN REGSAM samDesired,
351|     IN LPSECURITY_ATTRIBUTES lpSecurityAttributes,
352|     OUT PHKEY phkResult,
353|     OUT OPTIONAL LPDWORD lpdwDisposition
354| );
355|
356| typedef LONG
357| (WINAPI *tClusterRegOpenKey)(
358|     IN HKEY hKey,
359|     IN LPCWSTR lpszSubKey,
360|     IN REGSAM samDesired,
361|     OUT PHKEY phkResult
362| );
363|
364| typedef LONG
365| (WINAPI *tClusterRegDeleteKey)(

```

```

366|  IN HKEY hKey,
367|  IN LPCWSTR lpszSubKey
368|  );
369|
370| typedef LONG
371| (WINAPI *tClusterRegCloseKey)(
372|  IN HKEY hKey
373|  );
374|
375| typedef LONG
376| (WINAPI *tClusterRegEnumKey)(
377|  IN HKEY hKey,
378|  IN DWORD dwIndex,
379|  OUT LPWSTR lpszName,
380|  IN OUT LPDWORD lpcchName,
381|  OUT PFILETIME lpftLastWriteTime
382|  );
383|
384| typedef DWORD
385| (WINAPI *tClusterRegSetValue)(
386|  IN HKEY hKey,
387|  IN LPCWSTR lpszValueName,
388|  IN DWORD dwType,
389|  IN CONST BYTE* lpData,
390|  IN DWORD cbData
391|  );
392|
393| typedef DWORD
394| (WINAPI *tClusterRegDeleteValue)(
395|  IN HKEY hKey,
396|  IN LPCWSTR lpszValueName
397|  );
398|
399| typedef LONG
400| (WINAPI *tClusterRegQueryValue)(
401|  IN HKEY hKey,
402|  IN LPCWSTR lpszValueName,
403|  OUT LPDWORD lpdwValueType,
404|  OUT LPBYTE lpData,
405|  IN OUT LPDWORD lpcbData
406|  );
407|
408| typedef DWORD
409| (WINAPI *tClusterRegEnumValue)(
410|  IN HKEY hKey,
411|  IN DWORD dwIndex,
412|  OUT LPWSTR lpszValueName,
413|  IN OUT LPDWORD lpcchValueName,
414|  OUT LPDWORD lpdwType,
415|  OUT LPBYTE lpData,

```



```

416| IN OUT LPDWORD lpcbData
417| );
418|
419| typedef LONG
420| (WINAPI *tClusterRegQueryInfoKey)(
421| IN HKEY hKey,
422| IN LPDWORD lpcSubKeys,
423| IN LPDWORD lpcchMaxSubKeyLen,
424| IN LPDWORD lpcValues,
425| IN LPDWORD lpcchMaxValueNameLen,
426| IN LPDWORD lpcbMaxValueLen,
427| IN LPDWORD lpcbSecurityDescriptor,
428| IN PFILETIME lpftLastWriteTime
429| );
430|
431| typedef LONG
432| (WINAPI *tClusterRegGetKeySecurity )(
433| IN HKEY hKey,
434| IN SECURITY_INFORMATION RequestedInformation,
435| OUT PSECURITY_DESCRIPTOR pSecurityDescriptor,
436| IN LPDWORD lpcbSecurityDescriptor
437| );
438|
439| typedef LONG
440| (WINAPI *tClusterRegSetKeySecurity)(
441| IN HKEY hKey,
442| IN SECURITY_INFORMATION SecurityInformation,
443| IN PSECURITY_DESCRIPTOR pSecurityDescriptor
444| );
445|
446|
447| typedef DWORD
448| (WINAPI *tGetNodeClusterState)(
449| IN LPCWSTR lpszNodeName,
450| OUT DWORD *pdwClusterState
451| );
452|
453| typedef HCLUSTER
454| (WINAPI *tOpenCluster)(
455| IN LPCWSTR lpszClusterName
456| );
457|
458| typedef BOOL
459| (WINAPI *tCloseCluster)(
460| IN HCLUSTER hCluster
461| );
462|
463| typedef HKEY
464| (WINAPI *tGetClusterKey)(
465| IN HCLUSTER hCluster,

```

```

466| IN REGSAM samDesired
467| );
468|
469|
470| typedef struct sClusApis {
471|     HMODULE ClusApi;
472|     tOpenCluster OpenCluster;
473|     tCloseCluster CloseCluster;
474|     tGetClusterKey GetClusterKey;
475|     tClusterRegCreateKey ClusterRegCreateKey;
476|     tClusterRegOpenKey ClusterRegOpenKey;
477|     tClusterRegDeleteKey ClusterRegDeleteKey;
478|     tClusterRegCloseKey ClusterRegCloseKey;
479|     tClusterRegEnumKey ClusterRegEnumKey;
480|     tClusterRegSetValue ClusterRegSetValue;
481|     tClusterRegDeleteValue ClusterRegDeleteValue;
482|     tClusterRegQueryValue ClusterRegQueryValue;
483|     tClusterRegEnumValue ClusterRegEnumValue;
484|     tClusterRegQueryInfoKey ClusterRegQueryInfoKey;
485|     tClusterRegGetKeySecurity ClusterRegGetKeySecurity;
486|     tClusterRegSetKeySecurity ClusterRegSetKeySecurity;
487|     tGetNodeClusterState GetNodeClusterState;
488| } tClusApis,*pClusApis;
489|
490| ULONG UpdateClusterRegistries( tSnapShot *SnapShot );
491| ULONG IsClusterInstalled(void);
492| ULONG ClusterUpdateVolume( WCHAR *NTName );
493|
494|
495|
496| File Listing: event.c
497|
498| #include <windows.h>
499| #include <stdio.h>
500| #include <stdlib.h>
501| #include <process.h>
502| #include <tchar.h>
503| #include <winioctl.h>
504| #include <undoc.h>
505| #include "psm.h"
506|
507| #include "..\driver\ioctl.h"
508|
509| // this event is signalled when the
510| // service should end
511| //
512| extern HANDLE hServerStopEvent;
513| #define DLOG(x) printf x
514|
515| ULONG PSMI_OpenManager( HANDLE *hDevice )

```

```

516| {
517|     TCHAR Name[40]={0};
518|     ULONG returned=0;
519|     BOOL B=FALSE;
520|     ULONG Err=0;
521|
522|     | _sprintf(Name,TEXT("\\\\.\\%s_%04x"),TEXT("psman"),
523|     | PSM_LOW_COMPATIBLE_VERSION );
524|     *hDevice = CreateFile( Name,
525|     GENERIC_READ | GENERIC_WRITE,
526|     FILE_SHARE_READ | FILE_SHARE_WRITE,
527|     NULL,
528|     OPEN_EXISTING,
529|     FILE_FLAG_OVERLAPPED,
530|     NULL
531| );
532| if(*hDevice) {
533|     Err = 0;
534|
535| } else {
536|     Err = GetLastError();
537| }
538|
539| return Err;
540| }
541|
542| void DoVolumeOnlineEvent( void )
543| {
544|     return;
545| }
546|
547| ULONG DoCleanClusterDatabaseEvent( WCHAR *VolumeGuid,
548|     | WCHAR *UniqueId );
549|
550| void _cdecl PSMEventHandlerThread( void *Arg )
551| {
552|     HANDLE PSMAN;
553|     ULONG Err = PSMI_OpenManager(&PSMAN);
554|     if(!Err) {
555|         HANDLE Handles[2];
556|         OVERLAPPED o;
557|         tPSM_GetPSMEvent Event;
558|         ULONG returned;
559|
560|         Handles[0] = hServerStopEvent;
561|         o.hEvent = Handles[1] =
562|         | CreateEvent(NULL,TRUE,FALSE,NULL);

```

```

562|     while(1) {
563|         ResetEvent(Handles[1]);
564|
565|         // send command to driver
566|         | if(DeviceIoControl(PSMAN_IOCTL_GET_PSM_EVENT,NULL,0,&Event,
567|         | nt,sizeof(tPSM_GetPSMEvent),&returned,&o)) {
568|             Err = 0;
569|         } else {
570|             Err = GetLastError();
571|         }
572|         // we only ever expect io pending
573|         if((Err==ERROR_IO_PENDING) || (Err==0)) {
574|             // wait for an event to occur (or told
575|             | to exit)
576|             if(Err==0) {
577|                 Err = WAIT_OBJECT_0+1;
578|             } else {
579|                 Err =
580|                 | WaitForMultipleObjects(2,Handles,FALSE,INFINITE);
581|             }
582|             if(Err==WAIT_OBJECT_0) {
583|                 // told to exit
584|                 break;
585|             } else
586|             if(Err==WAIT_OBJECT_0+1) {
587|                 // driver specified event, find out
588|                 | which one
589|                 switch(Event.Event) {
590|                     case
591|                     | PSM_EVENT_CLEAN_CLUSTER_REGISTRY :
592|                     | DoCleanClusterDatabaseEvent( Event.VolumeGuid,
593|                     | Event.UniqueId );
594|                     break;
595|                     case PSM_EVENT_VOLUME_ONLINE:
596|                     | DLOG((TEXT("Volume '%S' is
597|                     | online\n"),Event.NTName));
598|                     | DoVolumeOnlineEvent();
599|                     break;
600|                     default:
601|                     | DLOG((TEXT("Unknown event
602|                     | %08x\n"),Event.Event));
603|                     // what to do???
604|                     break;
605|                 }
606|             } else {

```

```

602|          // what to do???
603|          DLOG((TEXT("Error %08x waiting for
| objects \n"),Err));
604|      }
605|  } else {
606|      DLOG((TEXT("Error %08x sending device
| control\n"),Err));
607|      if(Err==ERROR_INVALID_HANDLE) {
608|          // psm not installed
609|          // just exit
610|          break;
611|      }
612|      // what to do???
613|  }
614| } // while 1
615| CloseHandle(PSThread);
616| CloseHandle(Handles[1]);
617| } else {
618|     DLOG((TEXT("Error %08x opening
| manager\n"),Err));
619| }
620|
621| if ( hServerStopEvent ) {
622|     // set the event so the service will exit
623|     DLOG((TEXT("Setting stop event so service
| exits\n"))));
624|     SetEvent(hServerStopEvent);
625| }
626| }
627|
628|
629|
630| File Listing: main.c
631|
632| #include <windows.h>
633| #include <stdio.h>
634| #include <stdlib.h>
635| #include <process.h>
636| #include <tchar.h>
637| #include "service.h"
638|
639| // this event is signalled when the
640| // service should end
641| //
642| HANDLE hServerStopEvent = NULL;
643| extern void _cdecl PSMEventHandlerThread( void *Arg );
644|
645|
646| //
647| // FUNCTION: ServiceStart

```

```

648| //
649| // PURPOSE: Actual code of the service that does the
    | work.
650| //
651| // PARAMETERS:
652| // dwArgc - number of command line arguments
653| // lpszArgv - array of command line arguments
654| //
655| // RETURN VALUE:
656| // none
657| //
658| // COMMENTS:
659| //
660| VOID ServiceStart (DWORD dwArgc, LPTSTR *lpszArgv)
661| {
662|     HANDLE          hEvents[1] = {NULL};
663|     DWORD           dwWait;
664|
665|     //////////////////////////////////////
666|     //
667|     // Service initialization
668|     //
669|
670|     // report the status to the service control manager.
671|     //
672|     if (!ReportStatusToSCMgr(
673|         SERVICE_START_PENDING, //
        | service state
674|         NO_ERROR,              //
        | exit code
675|         3000))                 //
        | wait hint
676|         goto cleanup;
677|
678|     // create the event object. The control handler
        | function signals
679|     // this event when it receives the "stop" control
        | code.
680|     //
681|     hServerStopEvent = CreateEvent(
682|         NULL, // no
        | security attributes
683|         TRUE, // manual
        | reset event
684|         FALSE, //
        | not-signalled
685|         NULL); // no name
686|
687|     if ( hServerStopEvent == NULL)
688|         goto cleanup;

```

```

689|
690|  hEvents[0] = hServerStopEvent;
691|
692|  // create thread for event handler
693|  _beginthread(PSMEventHandlerThread,0,NULL);
694|
695|  // report the status to the service control manager.
696|  //
697|  if (!ReportStatusToSCMgr(
698|      SERVICE_START_PENDING, //
        | service state
699|      NO_ERROR,              //
        | exit code
700|      3000))                 //
        | wait hint
701|      goto cleanup;
702|
703|  //
704|  // End of initialization
705|  //
706|
        | //////////////////////////////////////
707|
708|  // report the status to the service control manager.
709|  //
710|  if (!ReportStatusToSCMgr(
711|      SERVICE_RUNNING,      //
        | service state
712|      NO_ERROR,              //
        | exit code
713|      0))                    //
        | wait hint
714|      goto cleanup;
715|
716|
        | //////////////////////////////////////
717|  //
718|  // Service is now running, perform work until
        | shutdown
719|  //
720|
721|  while ( 1 )
722|  {
723|      dwWait = WaitForMultipleObjects( 1, hEvents,
        | FALSE, INFINITE );
724|      if ( dwWait == WAIT_OBJECT_0 )
725|          break;
726|  }
727|
728|  cleanup:

```

```

729|
730|  if (hServerStopEvent)
731|      CloseHandle(hServerStopEvent);
732|
733| }
734|
735|
736| //
737| // FUNCTION: ServiceStop
738| //
739| // PURPOSE: Stops the service
740| //
741| // PARAMETERS:
742| //  none
743| //
744| // RETURN VALUE:
745| //  none
746| //
747| // COMMENTS:
748| //  If a ServiceStop procedure is going to
749| //  take longer than 3 seconds to execute,
750| //  it should spawn a thread to execute the
751| //  stop code, and return. Otherwise, the
752| //  ServiceControlManager will believe that
753| //  the service has stopped responding.
754| //
755| VOID ServiceStop()
756| {
757|  if ( hServerStopEvent )
758|      SetEvent(hServerStopEvent);
759| }
760|
761|
762|
763| File Listing: Service.c
764|
765| /*-----#
766|  | include <windows.h>
767| #include <stdio.h>
768| #include <stdlib.h>
769| #include <process.h>
770| #include <tchar.h>
771| #include <shellapi.h>
772| #include "service.h"
773|
774| // internal variables
775| SERVICE_STATUS      ssStatus;    // current
776| | status of the service
777| SERVICE_STATUS_HANDLE sshStatusHandle;

```



```

777| DWORD          dwErr = 0;
778| BOOL            bDebug = FALSE;
779| TCHAR           szErr[256];
780|
781| // internal function prototypes
782| VOID WINAPI service_ctrl(DWORD dwCtrlCode);
783| VOID WINAPI service_main(DWORD dwArgc, LPTSTR
    | *lpszArgv);
784| VOID CmdInstallService();
785| VOID CmdRemoveService();
786| VOID CmdDebugService(int argc, char **argv);
787| BOOL WINAPI ControlHandler ( DWORD dwCtrlType );
788| LPTSTR GetLastErrorText( LPTSTR lpszBuf, DWORD dwSize
    | );
789|
790| //
791| // FUNCTION: main
792| //
793| // PURPOSE: entrypoint for service
794| //
795| // PARAMETERS:
796| //  argc - number of command line arguments
797| //  argv - array of command line arguments
798| //
799| // RETURN VALUE:
800| //  none
801| //
802| // COMMENTS:
803| //  main() either performs the command line task, or
804| //  call StartServiceCtrlDispatcher to register the
805| //  main service thread. When the this call returns,
806| //  the service has stopped, so exit.
807| //
808| void __cdecl main(int argc, char **argv)
809| {
810|     SERVICE_TABLE_ENTRY dispatchTable[] =
811|     {
812|         { TEXT(SZSERVICENAME),
            | (LPSERVICE_MAIN_FUNCTION)service_main},
813|         { NULL, NULL}
814|     };
815|
816|     if ( (argc > 1) &&
817|         ((*argv[1] == '-') || (*argv[1] == '/')) )
818|     {
819|         if ( _stricmp( "install", argv[1]+1 ) == 0 )
820|         {
821|             CmdInstallService();
822|         }
823|         else if ( _stricmp( "remove", argv[1]+1 ) == 0 )

```

```

824|  {
825|      CmdRemoveService();
826|  }
827|  else if ( _stricmp( "debug", argv[1]+1 ) == 0 )
828|  {
829|      bDebug = TRUE;
830|      CmdDebugService(argc, argv);
831|  }
832|  else
833|  {
834|      goto dispatch;
835|  }
836|  exit(0);
837| }
838|
839| // if it doesn't match any of the above parameters
840| // the service control manager may be starting the
    | service
841| // so we must call StartServiceCtrlDispatcher
842| dispatch:
843| // this is just to be friendly
844| printf( "%s -install      to install the
    | service\n", SZAPPNAME );
845| printf( "%s -remove      to remove the
    | service\n", SZAPPNAME );
846| printf( "%s -debug <params> to run as a console
    | app for debugging\n", SZAPPNAME );
847| printf( "\nStartServiceCtrlDispatcher being
    | called.\n" );
848| printf( "This may take several seconds. Please
    | wait.\n" );
849|
850| if (!StartServiceCtrlDispatcher(dispatchTable))
851|     AddToMessageLog(TEXT("StartServiceCtrlDispatcher
    | failed."));
852| }
853|
854|
855|
856| //
857| // FUNCTION: service_main
858| //
859| // PURPOSE: To perform actual initialization of the
    | service
860| //
861| // PARAMETERS:
862| // dwArgc - number of command line arguments
863| // lpszArgv - array of command line arguments
864| //
865| // RETURN VALUE:

```

```

866| // none
867| //
868| // COMMENTS:
869| // This routine performs the service initialization
    | and then calls
870| // the user defined ServiceStart() routine to
    | perform majority
871| // of the work.
872| //
873| void WINAPI service_main(DWORD dwArgc, LPTSTR
    | *lpszArgv)
874| {
875|
876| // register our service control handler:
877| //
878| sshStatusHandle = RegisterServiceCtrlHandler(
    | TEXT(SZSERVICENAME), service_ctrl);
879|
880| if (!sshStatusHandle)
881|     goto cleanup;
882|
883| // SERVICE_STATUS members that don't change in
    | example
884| //
885| ssStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
886| ssStatus.dwServiceSpecificExitCode = 0;
887|
888|
889| // report the status to the service control manager.
890| //
891| if (!ReportStatusToSCMGr(
892|     SERVICE_START_PENDING, //
    | service state
893|     NO_ERROR,              //
    | exit code
894|     3000))                 //
    | wait hint
895|     goto cleanup;
896|
897|
898| ServiceStart( dwArgc, lpszArgv );
899|
900| cleanup:
901|
902| // try to report the stopped status to the service
    | control manager.
903| //
904| if (sshStatusHandle)
905|     (VOID)ReportStatusToSCMGr(
906|         SERVICE_STOPPED,

```

```

907|             dwErr,
908|             0);
909|
910|     return;
911| }
912|
913|
914|
915| //
916| // FUNCTION: service_ctrl
917| //
918| // PURPOSE: This function is called by the SCM
919| | whenever
920| | ControlService() is called on this
921| | service.
922| //
923| // PARAMETERS:
924| // dwCtrlCode - type of control requested
925| //
926| // RETURN VALUE:
927| // none
928| //
929| // COMMENTS:
930| //
931| VOID WINAPI service_ctrl(DWORD dwCtrlCode)
932| {
933|     // Handle the requested control code.
934|     //
935|     switch (dwCtrlCode)
936|     {
937|         // Stop the service.
938|         //
939|         // SERVICE_STOP_PENDING should be reported before
940|         // setting the Stop Event - hServerStopEvent - in
941|         // ServiceStop(). This avoids a race condition
942|         // which may result in a 1053 - The Service did not
943|         // respond...
944|         // error.
945|         case SERVICE_CONTROL_STOP:
946|             ReportStatusToSCMgr(SERVICE_STOP_PENDING,
947|             | NO_ERROR, 0);
948|             ServiceStop();
949|             return;
950|
951|         // Update the service status.
952|         //
953|         case SERVICE_CONTROL_INTERROGATE:
954|             break;
955|
956|         // invalid control code

```

```

953|    //
954|    default:
955|        break;
956|
957|    }
958|
959|    ReportStatusToSCMgr(ssStatus.dwCurrentState,
    | NO_ERROR, 0);
960| }
961|
962|
963|
964| //
965| // FUNCTION: ReportStatusToSCMgr()
966| //
967| // PURPOSE: Sets the current status of the service and
968| //          reports it to the Service Control Manager
969| //
970| // PARAMETERS:
971| //   dwCurrentState - the state of the service
972| //   dwWin32ExitCode - error code to report
973| //   dwWaitHint - worst case estimate to next
    | checkpoint
974| //
975| // RETURN VALUE:
976| //   TRUE - success
977| //   FALSE - failure
978| //
979| // COMMENTS:
980| //
981| BOOL ReportStatusToSCMgr(DWORD dwCurrentState,
982|                          DWORD dwWin32ExitCode,
983|                          DWORD dwWaitHint)
984| {
985|     static DWORD dwCheckPoint = 1;
986|     BOOL fResult = TRUE;
987|
988|
989|     if ( !bDebug ) // when debugging we don't report to
    | the SCM
990|     {
991|         if (dwCurrentState == SERVICE_START_PENDING)
992|             ssStatus.dwControlsAccepted = 0;
993|         else
994|             ssStatus.dwControlsAccepted =
    | SERVICE_ACCEPT_STOP;
995|
996|         ssStatus.dwCurrentState = dwCurrentState;
997|         ssStatus.dwWin32ExitCode = dwWin32ExitCode;
998|         ssStatus.dwWaitHint = dwWaitHint;

```

```

999|
1000|     if ( ( dwCurrentState == SERVICE_RUNNING ) ||
1001|         ( dwCurrentState == SERVICE_STOPPED ) )
1002|         ssStatus.dwCheckPoint = 0;
1003|     else
1004|         ssStatus.dwCheckPoint = dwCheckPoint++;
1005|
1006|
1007|     // Report the status of the service to the
        | service control manager.
1008|     //
1009|     if (!fResult = SetServiceStatus(
        | sshStatusHandle, &ssStatus)))
1010|     {
1011|         AddToMessageLog(TEXT("SetServiceStatus"));
1012|     }
1013| }
1014| return fResult;
1015| }
1016|
1017|
1018|
1019| //
1020| // FUNCTION: AddToMessageLog(LPTSTR lpszMsg)
1021| //
1022| // PURPOSE: Allows any thread to log an error message
1023| //
1024| // PARAMETERS:
1025| //     lpszMsg - text for message
1026| //
1027| // RETURN VALUE:
1028| //     none
1029| //
1030| // COMMENTS:
1031| //
1032| VOID AddToMessageLog(LPTSTR lpszMsg)
1033| {
1034| #if 0
1035|     TCHAR  szMsg[256];
1036|     HANDLE hEventSource;
1037|     LPTSTR lpszStrings[2];
1038|
1039|
1040|     if ( !bDebug )
1041|     {
1042|         dwErr = GetLastError();
1043|
1044|         // Use event logging to log the error.
1045|         //
1046|         hEventSource = RegisterEventSource(NULL,

```

```

    | TEXT(SZSERVICENAME));
1047|
1048|     _sprintf(szMsg, TEXT("%s error: %d"),
    | TEXT(SZSERVICENAME), dwErr);
1049|     lpszStrings[0] = szMsg;
1050|     lpszStrings[1] = lpszMsg;
1051|
1052|     if (hEventSource != NULL)
1053|     {
1054|         ReportEvent(hEventSource, // handle of event
    | source
1055|             EVENTLOG_ERROR_TYPE, // event
    | type
1056|             0,                    // event
    | category
1057|             0,                    // event ID
1058|             NULL,                 // current
    | user's SID
1059|             2,                    // strings
    | in lpszStrings
1060|             0,                    // no bytes
    | of raw data
1061|             lpszStrings,          // array of
    | error strings
1062|             NULL);                // no raw
    | data
1063|
1064|     (VOID) DeregisterEventSource(hEventSource);
1065|     }
1066| }
1067| #endif
1068| }
1069|
1070|
1071|
1072|
1073| //////////////////////////////////////
    | //////////
1074| //
1075| // The following code handles service installation and
    | removal
1076| //
1077|
1078|
1079| //
1080| // FUNCTION: CmdInstallService()
1081| //
1082| // PURPOSE: Installs the service
1083| //
1084| // PARAMETERS:

```

```

1085| // none
1086| //
1087| // RETURN VALUE:
1088| // none
1089| //
1090| // COMMENTS:
1091| //
1092| void CmdInstallService()
1093| {
1094|     SC_HANDLE schService;
1095|     SC_HANDLE schSCManager;
1096|
1097|     TCHAR szPath[512];
1098|
1099|     if ( GetModuleFileName( NULL, szPath, 512 ) == 0 )
1100|     {
1101|         _tprintf(TEXT("Unable to install %s - %s\n"),
1102|             | TEXT(SZSERVICEDISPLAYNAME), GetLastErrorText(szErr,
1103|             | 256));
1104|         return;
1105|     }
1106|
1107|     schSCManager = OpenSCManager(
1108|         NULL,
1109|         | // machine (NULL == local)
1110|         NULL,
1111|         | // database (NULL == default)
1112|         SC_MANAGER_ALL_ACCESS
1113|         | // access required
1114|     );
1115|     if ( schSCManager )
1116|     {
1117|         schService = CreateService(
1118|             schSCManager,
1119|             | // SCManager database
1120|             TEXT(SZSERVICENAME),
1121|             | // name of service
1122|             | TEXT(SZSERVICEDISPLAYNAME), // name to display
1123|             SERVICE_ALL_ACCESS,
1124|             | // desired access
1125|             SERVICE_WIN32_OWN_PROCESS, // service type
1126|             SERVICE_AUTO_START,
1127|             | // start type
1128|             SERVICE_ERROR_NORMAL,
1129|             | // error control type
1130|             szPath,
1131|             | // service's binary
1132|             NULL,

```



```

    | // no load ordering group
1122|             NULL,
    | // no tag identifier
1123|             TEXT(SZDEPENDENCIES),
    | // dependencies
1124|             NULL,
    | // LocalSystem account
1125|             NULL);
    | // no password
1126|
1127|     if ( schService )
1128|     {
1129|         _tprintf(TEXT("%s installed.\n"),
    | TEXT(SZSERVICEDISPLAYNAME) );
1130|         CloseServiceHandle(schService);
1131|     }
1132|     else
1133|     {
1134|         _tprintf(TEXT("CreateService failed - %s\n"),
    | GetLastErrorText(szErr, 256));
1135|     }
1136|
1137|     CloseServiceHandle(schSCManager);
1138| }
1139| else
1140|     _tprintf(TEXT("OpenSCManager failed - %s\n"),
    | GetLastErrorText(szErr,256));
1141| }
1142|
1143|
1144|
1145| //
1146| // FUNCTION: CmdRemoveService()
1147| //
1148| // PURPOSE: Stops and removes the service
1149| //
1150| // PARAMETERS:
1151| //     none
1152| //
1153| // RETURN VALUE:
1154| //     none
1155| //
1156| // COMMENTS:
1157| //
1158| void CmdRemoveService()
1159| {
1160|     SC_HANDLE  schService;
1161|     SC_HANDLE  schSCManager;
1162|
1163|     schSCManager = OpenSCManager(

```

```

1164|             NULL,
    | // machine (NULL == local)
1165|             NULL,
    | // database (NULL == default)
1166|             SC_MANAGER_ALL_ACCESS
    | // access required
1167|         );
1168|     if ( schSCManager )
1169|     {
1170|         schService = OpenService(schSCManager,
    | TEXT(SZSERVICENAME), SERVICE_ALL_ACCESS);
1171|
1172|         if (schService)
1173|         {
1174|             // try to stop the service
1175|             if ( ControlService( schService,
    | SERVICE_CONTROL_STOP, &ssStatus ) )
1176|             {
1177|                 _tprintf(TEXT("Stopping %s."),
    | TEXT(SZSERVICEDISPLAYNAME));
1178|                 Sleep( 1000 );
1179|
1180|                 while ( QueryServiceStatus( schService,
    | &ssStatus ) )
1181|                 {
1182|                     if ( ssStatus.dwCurrentState ==
    | SERVICE_STOP_PENDING )
1183|                     {
1184|                         _tprintf(TEXT("."));
1185|                         Sleep( 1000 );
1186|                     }
1187|                     else
1188|                         break;
1189|                 }
1190|
1191|                 if ( ssStatus.dwCurrentState ==
    | SERVICE_STOPPED )
1192|                     _tprintf(TEXT("\n%s stopped.\n"),
    | TEXT(SZSERVICEDISPLAYNAME) );
1193|                 else
1194|                     _tprintf(TEXT("\n%s failed to stop.\n"),
    | TEXT(SZSERVICEDISPLAYNAME) );
1195|
1196|             }
1197|
1198|             // now remove the service
1199|             if ( DeleteService(schService) )
1200|                 _tprintf(TEXT("%s removed.\n"),
    | TEXT(SZSERVICEDISPLAYNAME) );
1201|             else

```

```

1202|         _tprintf(TEXT("DeleteService failed -
| %s\n"), GetLastErrorText(szErr,256));
1203|
1204|
1205|         CloseServiceHandle(schService);
1206|     }
1207|     else
1208|         _tprintf(TEXT("OpenService failed - %s\n"),
| GetLastErrorText(szErr,256));
1209|
1210|     CloseServiceHandle(schSCManager);
1211| }
1212| else
1213|     _tprintf(TEXT("OpenSCManager failed - %s\n"),
| GetLastErrorText(szErr,256));
1214| }
1215|
1216|
1217|
1218|
1219| //////////////////////////////////////
| //////////////////////////////////
1220| //
1221| // The following code is for running the service as a
| console app
1222| //
1223|
1224|
1225| //
1226| // FUNCTION: CmdDebugService(int argc, char ** argv)
1227| //
1228| // PURPOSE: Runs the service as a console application
1229| //
1230| // PARAMETERS:
1231| //   argc - number of command line arguments
1232| //   argv - array of command line arguments
1233| //
1234| // RETURN VALUE:
1235| //   none
1236| //
1237| // COMMENTS:
1238| //
1239| void CmdDebugService(int argc, char ** argv)
1240| {
1241|     DWORD dwArgc;
1242|     LPTSTR *lpszArgv;
1243|
1244| #ifdef UNICODE
1245|     lpszArgv = CommandLineToArgvW(GetCommandLineW(),
| &(dwArgc) );

```

```

1246| #else
1247|  dwArgc  = (DWORD) argc;
1248|  lpszArgv = argv;
1249| #endif
1250|
1251|  _tprintf(TEXT("Debugging %s.\n"),
    | TEXT(SZSERVICEDISPLAYNAME));
1252|
1253|  SetConsoleCtrlHandler( ControlHandler, TRUE );
1254|
1255|  ServiceStart( dwArgc, lpszArgv );
1256|
1257| #ifdef UNICODE
1258| // Must free memory allocated for arguments
1259|
1260|  GlobalFree(lpszArgv);
1261| #endif // UNICODE
1262|
1263| }
1264|
1265|
1266| //
1267| // FUNCTION: ControlHandler ( DWORD dwCtrlType )
1268| //
1269| // PURPOSE: Handled console control events
1270| //
1271| // PARAMETERS:
1272| //  dwCtrlType - type of control event
1273| //
1274| // RETURN VALUE:
1275| //  True - handled
1276| //  False - unhandled
1277| //
1278| // COMMENTS:
1279| //
1280| BOOL WINAPI ControlHandler ( DWORD dwCtrlType )
1281| {
1282|  switch ( dwCtrlType )
1283|  {
1284|  case CTRL_BREAK_EVENT: // use Ctrl+C or Ctrl+Break
    | to simulate
1285|  case CTRL_C_EVENT:    // SERVICE_CONTROL_STOP in
    | debug mode
1286|    _tprintf(TEXT("Stopping %s.\n"),
    | TEXT(SZSERVICEDISPLAYNAME));
1287|    ServiceStop();
1288|    return TRUE;
1289|    break;
1290|
1291|  }

```

```

1292|  return FALSE;
1293| }
1294|
1295| //
1296| // FUNCTION: GetLastErrorText
1297| //
1298| // PURPOSE: copies error message text to string
1299| //
1300| // PARAMETERS:
1301| //  lpszBuf - destination buffer
1302| //  dwSize - size of buffer
1303| //
1304| // RETURN VALUE:
1305| //  destination buffer
1306| //
1307| // COMMENTS:
1308| //
1309| LPTSTR GetLastErrorText( LPTSTR lpszBuf, DWORD dwSize )
1310| {
1311|  DWORD dwRet;
1312|  LPTSTR lpszTemp = NULL;
1313|
1314|  dwRet = FormatMessage(
    | FORMAT_MESSAGE_ALLOCATE_BUFFER |
    | FORMAT_MESSAGE_FROM_SYSTEM
    | |FORMAT_MESSAGE_ARGUMENT_ARRAY,
1315|      NULL,
1316|      GetLastError(),
1317|      LANG_NEUTRAL,
1318|      (LPTSTR)&lpszTemp,
1319|      0,
1320|      NULL );
1321|
1322|  // supplied buffer is not long enough
1323|  if ( !dwRet || ( (long)dwSize < (long)dwRet+14 ) )
1324|      lpszBuf[0] = TEXT('\0');
1325|  else
1326|  {
1327|      lpszTemp[lstrlen(lpszTemp)-2] = TEXT('\0');
    | //remove cr and newline character
1328|      _stprintf( lpszBuf, TEXT("%s (0x%x)", lpszTemp,
    | GetLastError() );
1329|  }
1330|
1331|  if ( lpszTemp )
1332|      LocalFree((HLOCAL) lpszTemp );
1333|
1334|  return lpszBuf;
1335| }
1336|

```

```

1337|
1338|
1339| File Listing: Service.h
1340|
1341| #ifndef _SERVICE_H
1342| #define _SERVICE_H
1343|
1344|
1345| #ifdef __cplusplus
1346| extern "C" {
1347| #endif
1348|
1349|
1350| //////////////////////////////////////
| //////////////////////////////////
1351| /// todo: change to desired strings
1352| ///
1353| // name of the executable
1354| #define SZAPPNAME          "psmserv"
1355| // internal name of the service
1356| #define SZSERVICENAME      "psmserv"
1357| // displayed name of the service
1358| #define SZSERVICEDISPLAYNAME "Persistent Storage
| Manager Service"
1359| // list of service dependencies - "dep1\0dep2\0\0"
1360| #define SZDEPENDENCIES     ""
1361| //////////////////////////////////////
| //////////////////////////////////
1362|
1363|
1364|
1365| //////////////////////////////////////
| //////////////////////////////////
1366| /// todo: ServiceStart() must be defined by in your
| code.
1367| /// The service should use ReportStatusToSCMgr
| to indicate
1368| /// progress. This routine must also be used by
| StartService()
1369| /// to report to the SCM when the service is
| running.
1370| ///
1371| /// If a ServiceStop procedure is going to take
| longer than
1372| /// 3 seconds to execute, it should spawn a
| thread to
1373| /// execute the stop code, and return.
| Otherwise, the
1374| /// ServiceControlManager will believe that the
| service has

```

```

1375| ///    stopped responding
1376| ///
1377| VOID ServiceStart(DWORD dwArgc, LPTSTR *lpszArgv);
1378| VOID ServiceStop();
1379| //////////////////////////////////////
    | //////////////////////////////////
1380|
1381|
1382|
1383| //////////////////////////////////////
    | //////////////////////////////////
1384| /// The following are procedures which
1385| /// may be useful to call within the above procedures,
1386| /// but require no implementation by the user.
1387| /// They are implemented in service.c
1388|
1389| //
1390| // FUNCTION: ReportStatusToSCMgr()
1391| //
1392| // PURPOSE: Sets the current status of the service and
1393| //          reports it to the Service Control Manager
1394| //
1395| // PARAMETERS:
1396| //   dwCurrentState - the state of the service
1397| //   dwWin32ExitCode - error code to report
1398| //   dwWaitHint - worst case estimate to next
    | checkpoint
1399| //
1400| // RETURN VALUE:
1401| //   TRUE - success
1402| //   FALSE - failure
1403| //
1404| BOOL ReportStatusToSCMgr(DWORD dwCurrentState, DWORD
    | dwWin32ExitCode, DWORD dwWaitHint);
1405|
1406|
1407| //
1408| // FUNCTION: AddToMessageLog(LPTSTR lpszMsg)
1409| //
1410| // PURPOSE: Allows any thread to log an error message
1411| //
1412| // PARAMETERS:
1413| //   lpszMsg - text for message
1414| //
1415| // RETURN VALUE:
1416| //   none
1417| //
1418| void AddToMessageLog(LPTSTR lpszMsg);
1419| //////////////////////////////////////
    | //////////////////////////////////

```

```

1420|
1421|
1422| #ifdef __cplusplus
1423| }
1424| #endif
1425|
1426| #endif
1427|
1428|
1429|
1430| PSM Command Line Source
1431|
1432| The Command Line allows the user and the scheduling
1433| system to issue commands to the PSM system. --LPW
1434|
1435|
1436|
1437| File Listing: File: bootini.c
1438|
1439| #include "precomp.h"
1440|
1441| //-----
| -----
| --
1442|
1443| STATIC ULONG UpdateBootIniBuffer (
1444|     const char    *OriginalBuffer,
1445|     const ULONG    OriginalBytesRead,
1446|     char          *TempBuffer,
1447|     ULONG          *TempBytes,
1448|     const ULONG    TempBufferSize );
1449|
1450| //-----
| -----
| --
1451|
1452| void UpdateBootIni()
1453| {
1454|     //-----
| -----
1455|     // This function determines whether the file
| '%systemdrive%\boot.ini' should
1456|     // be copied and updated in
| '%systemroot%\system32\boot.ini'.
1457|     // The purpose is to make sure we always have
| '/nopsm' and '/resetpsm' options
1458|     // available if someone boots up Recovery Console,
| so they can manually copy
1459|     // boot.ini back into the root of the boot drive.

```



```

1460|  //
1461|  // We avoid re-creating the copy if the copy exists
    | already and its contents
1462|  // are identical to what we would replace it with.
1463|  // The rationale for this is that if the computer
    | crashes soon after we
1464|  // overwrite the copy of boot.ini, it may not be
    | flushed from the cache manager
1465|  // and not be available. Writing only when
    | necessary greatly reduces this risk.
1466|  //
1467|  // However, we don't just avoid overwriting the
    | copy just because already exists.
1468|  // This is because the system administrator may
    | edit the original boot.ini at any
1469|  // time, and we want to get a freshly updated copy.
1470|  //
1471|  // We also need to detect the case where the
    | updated copy has been manually
1472|  // copied to the root, so we don't keep adding
    | superfluous boot entries.
1473|
    | //-----
    | -----
1474|
1475|  const ULONG BUFFER_SIZE = 64 * 1024;
1476|  const ULONG MAX_BOOTINI_SIZE = BUFFER_SIZE / 4;
1477|  char *OriginalBuffer = NULL;
1478|  char *UpdatedBuffer = NULL;
1479|  char *TempBuffer = NULL;
1480|  FILE *OriginalFile = NULL;
1481|  FILE *UpdatedFile = NULL;
1482|  const char *Env = NULL;
1483|  char OriginalFileName [256];
1484|  char UpdatedFileName [256];
1485|  char BackupFileName [256];
1486|
1487|  DEBUG_WRITE(L">>> Entering UpdateBootIni()\n");
1488|
1489|  Env = getenv("SystemDrive");
1490|  if ( !Env ) {
1491|      DEBUG_WRITE(L">>> !!! Environment variable
    | 'SystemDrive' not found !!!\n");
1492|  } else {
1493|      sprintf (OriginalFileName, "%s\\boot.ini",
    | Env);
1494|      DEBUG_WRITE(L">>> OriginalFileName = '%S'\n",
    | OriginalFileName);
1495|      Env = getenv("SystemRoot");
1496|      if ( !Env ) {

```

```

1497|         DEBUG_WRITE(L">>> !!! Environment variable
    | 'SystemRoot' not found!!!\n");
1498|     } else {
1499|         sprintf (UpdatedFileName,
    | "%s\\system32\\boot.ini", Env);
1500|         DEBUG_WRITE(L">>> UpdatedFileName =
    | '%S'\n", UpdatedFileName);
1501|
1502|         sprintf (BackupFileName,
    | "%s\\system32\\boot.sav", Env);
1503|         DEBUG_WRITE(L">>> BackupFileName =
    | '%S'\n", BackupFileName);
1504|
1505|         OriginalBuffer = (char *)
    | malloc(BUFFER_SIZE);
1506|         if ( !OriginalBuffer ) {
1507|             WRITE_ERROR(L"Warning: Could not
    | allocate memory for boot.ini safe copy (1)\n");
1508|         } else {
1509|             memset (OriginalBuffer, 0,
    | BUFFER_SIZE);
1510|             UpdatedBuffer = (char *)
    | malloc(BUFFER_SIZE);
1511|             if ( !UpdatedBuffer ) {
1512|                 WRITE_ERROR(L"Warning: Could not
    | allocate memory for boot.ini safe copy (2)\n");
1513|             } else {
1514|                 memset (UpdatedBuffer, 0,
    | BUFFER_SIZE);
1515|                 TempBuffer = (char *)
    | malloc(BUFFER_SIZE);
1516|                 if ( !TempBuffer ) {
1517|                     WRITE_ERROR(L"Warning: Could
    | not allocate memory for boot.ini safe copy (3)\n");
1518|                 } else {
1519|                     memset (TempBuffer, 0,
    | BUFFER_SIZE);
1520|                     OriginalFile =
    | fopen(OriginalFileName,"rt");
1521|                     if ( !OriginalFile ) {
1522|                         DEBUG_WRITE(L"Cannot open
    | file '%S' for read\n",OriginalFileName);
1523|                     } else {
1524|                         ULONG OriginalBytesRead =
    | fread (OriginalBuffer, 1, BUFFER_SIZE, OriginalFile);
1525|                         fclose(OriginalFile);
1526|                         OriginalFile = NULL;
1527|                         DEBUG_WRITE(L">>> Read %u
    | bytes from '%S'\n",OriginalBytesRead,OriginalFileName);
1528|

```

```

1529|             if ( OriginalBytesRead >=
| MAX_BOOTINI_SIZE ) {
1530|                 WRITE_ERROR(L"Warning:
| File '%S' is too big to make safe
| copy\n",OriginalFileName);
1531|             } else {
1532|                 ULONG UpdateSuccess =
| FALSE;
1533|                 ULONG UpdatedBytesRead
| = 0;
1534|                 ULONG TempBytes = 0;
| // number of bytes generated in TempBuffer
1535|                 ULONG ShouldMakeCopy =
| FALSE;
1536|
1537|                 UpdatedFile =
| fopen(UpdatedFileName,"rt");
1538|                 if ( !UpdatedFile ) {
1539|                     ShouldMakeCopy =
| TRUE;
1540|                     DEBUG_WRITE(L">>>
| UpdateBootIni: Updated file does not already exist\n");
1541|                 } else {
1542|                     UpdatedBytesRead =
| fread(UpdatedBuffer, 1, BUFFER_SIZE, UpdatedFile);
1543|                     fclose
| (UpdatedFile);
1544|                     UpdatedFile = NULL;
1545|                     DEBUG_WRITE(L">>>
| Read %u bytes from
| '%S'\n",UpdatedBytesRead,UpdatedFileName);
1546|                 }
1547|
1548|
| OriginalBuffer[OriginalBytesRead] = '\0';
1549|
| UpdatedBuffer[UpdatedBytesRead] = '\0';
1550|
1551|                 // Generate updated
| boot.ini image in TempBuffer based on OriginalBuffer.
1552|                 // Then compare to
| UpdatedBuffer. If not identical, then overwrite
| updated
1553|                 // file with
| TempBuffer.
1554|
1555|                 UpdateSuccess =
| UpdateBootIniBuffer (
1556|                     OriginalBuffer,
1557|                     OriginalBytesRead,

```

```

1558|             TempBuffer,
1559|             &TempBytes,
1560|             BUFFER_SIZE );
1561|
1562|             if ( UpdateSuccess &&
| !ShouldMakeCopy ) {
1563|                 if (
| TempBytes!=UpdatedBytesRead ||
| memcmp(TempBuffer,UpdatedBuffer,UpdatedBytesRead)!=0 )
| {
1564|                     | DEBUG_WRITE(L">>> Contents of '%S' do not match
| in-memory updated image.\n",UpdatedFileName);
1565|                     ShouldMakeCopy
| = TRUE;
1566|                 }
1567|             }
1568|
1569|             if ( UpdateSuccess &&
| ShouldMakeCopy ) {
1570|                 FILE *BackupFile =
| fopen (BackupFileName, "wt");
1571|                 if ( BackupFile ) {
1572|                     ULONG
| BackupBytesWritten = fwrite (OriginalBuffer, 1,
| OriginalBytesRead, BackupFile);
1573|                     fclose
| (BackupFile);
1574|                     BackupFile =
| NULL;
1575|                     if (
| BackupBytesWritten == OriginalBytesRead ) {
1576|                         | DEBUG_WRITE(L">>> Successfully made backup of '%S' in
| '%S'\n", OriginalFileName, BackupFileName);
1577|                     } else {
1578|                         | DEBUG_WRITE(L">>> !!! Could only write %u bytes to
| '%S'\n", BackupFileName);
1579|                     }
1580|                 } else {
1581|                     | DEBUG_WRITE(L">>> Could not make backup file '%S'\n",
| BackupFileName);
1582|                 }
1583|
1584|                 DEBUG_WRITE(L">>>
| **** Trying to update '%S'\n",UpdatedFileName);
1585|                 UpdatedFile = fopen
| (UpdatedFileName, "wt");

```

```

1586|                if ( !UpdatedFile )
1587|                | {
1588|                | WRITE_ERROR(L"Warning: Could not open '%S' to write
1589|                | safe copy of '%S'\n",UpdatedFileName,OriginalFileName);
1590|                | } else {
1591|                |     ULONG
1592|                |     UpdatedBytesWritten = fwrite (TempBuffer, 1, TempBytes,
1593|                |     UpdatedFile);
1594|                |     fclose
1595|                |     (UpdatedFile);
1596|                |     UpdatedFile =
1597|                |     NULL;
1598|                |
1599|                |     DEBUG_WRITE(L">>> Wrote %u bytes to
1600|                |     '%S'\n",UpdatedBytesWritten,UpdatedFileName);
1601|                |     if (
1602|                |     UpdatedBytesWritten == TempBytes ) {
1603|                |     DEBUG_WRITE(L">>> **** Safe copy of boot.ini was
1604|                |     successful!\n");
1605|                |     } else {
1606|                |     WRITE_ERROR(L"Warning: Could not write complete safe
1607|                |     copy of '%S' to
1608|                |     '%S'\n",OriginalFileName,UpdatedFileName);
1609|                |     }
1610|                |     }
1611|                |     } else {
1612|                |     DEBUG_WRITE(L">>>
1613|                |     **** Leaving alone existing '%S'\n",UpdatedFileName);
1614|                |     }
1615|                |     }
1616|                |     }
1617|                |     }
1618|                |     }
1619|

```

```

1620| //-----
    | -----
    | --
1621|
1622| STATIC ULONG ExtractLine (
1623|     const char *InBuffer,
1624|     char      *LineBuffer,
1625|     const ULONG LineBufferSize,
1626|     ULONG      *LineLength,
1627|     ULONG      *IsLastLine )
1628| {
1629|     ULONG Success = FALSE;
1630|
1631|     if ( LineLength!=NULL && LineBuffer!=NULL &&
        | InBuffer!=NULL && LineBufferSize>0 && IsLastLine!=NULL
        | ) {
1632|         *LineLength = 0;
1633|         *IsLastLine = FALSE;
1634|
1635|         while ( InBuffer[*LineLength]!='\0' &&
            | InBuffer[*LineLength]!='\n' &&
            | *LineLength<LineBufferSize ) {
1636|             LineBuffer[*LineLength] =
            | InBuffer[*LineLength];
1637|             ++(*LineLength);
1638|         }
1639|
1640|         if ( *LineLength >= LineBufferSize ) {
1641|             WRITE_ERROR(L">>> Line overflow in
            | ExtractLine\n");
1642|         } else {
1643|             if ( InBuffer[*LineLength] == '\0' ) {
1644|                 *IsLastLine = TRUE;
1645|             }
1646|             LineBuffer[*LineLength] = '\0';
1647|             Success = TRUE;
1648|         }
1649|     } else {
1650|         WRITE_ERROR(L">>> Internal Error: Invalid
            | parameter in ExtractLine\n");
1651|     }
1652|
1653|     return Success;
1654| }
1655|
1656| //-----
    | -----
    | --
1657|
1658| STATIC ULONG UpdateBootIniBuffer (

```



```

| (
1705|         &OriginalBuffer[OriginalIndex],
1706|         OriginalLine,
1707|         sizeof(OriginalLine),
1708|         &LineLength,
1709|         &IsLastLine );
1710|
1711|         if ( !ExtractSuccess ) {
1712|             Success = FALSE;
1713|             break;
1714|         }
1715|
1716|         // See if we are changing sections.
1717|         if ( OriginalLine[0] == '[' ) {
1718|             InOperatingSystemsSection =
| (strcmp(OriginalLine,"[operating systems]") == 0);
1719|         } else if (
| InOperatingSystemsSection ) {
1720|             // Look for any lines in
| operating system section already containing known
| switches
1721|
1722|             for ( i=0;
| PsmSwitches[i].Switch != NULL; ++i ) {
1723|                 if (
| strstr(OriginalLine,PsmSwitches[i].Switch) ) {
1724|                     ShouldCopyThisLine =
| FALSE;
1725|                     break;
1726|                 }
1727|             }
1728|
1729|             for ( i=0; ExcludedSwitches[i]
| != NULL; ++i ) {
1730|                 if (
| strstr(OriginalLine,ExcludedSwitches[i]) ) {
1731|                     ShouldExpandThisLine =
| FALSE;
1732|                     break;
1733|                 }
1734|             }
1735|         }
1736|
1737|         if ( !IsLastLine ) {
1738|             strcat ( OriginalLine, "\n" );
1739|             ++LineLength;
1740|         }
1741|
1742|         if ( ShouldCopyThisLine ) {
1743|             if ( *TempBytes + LineLength >=

```



```

    | TempBufferSize ) {
1744|         Success = FALSE;
1745|         WRITE_ERROR(L"Error: Safe
    | copy of boot.ini is too long! (1)\n");
1746|         break;
1747|     }
1748|
1749|     strcpy (
    | &TempBuffer[*TempBytes], OriginalLine );
1750|     *TempBytes += LineLength;
1751|
1752|     // Now see if we need to make
    | modified duplicates of this line...
1753|     if ( InOperatingSystemsSection
    | && ShouldExpandThisLine ) {
1754|         const char *EqualSign =
    | strchr(OriginalLine, '=');
1755|         if ( EqualSign ) {
1756|             const char *FirstQuote
    | = strchr(EqualSign+1, "\"");
1757|             if ( FirstQuote ) {
1758|                 const char
    | *SecondQuote = strchr(FirstQuote+1, "\"");
1759|                 if ( SecondQuote )
    | {
1760|                     // Expect a
    | line like 'multi(0)disk(0)...\WINNT="Microsoft Windows
    | ..." /stuff'
1761|
1762|                     for ( i=0;
    | PsmSwitches[i].Switch != NULL; ++i ) {
1763|                         const char
    | *pIn = OriginalLine;
1764|                         const char
    | *pDesc = PsmSwitches[i].Description;
1765|                         const char
    | *pSwitch = PsmSwitches[i].Switch;
1766|                         char *pOut
    | = DuplicateLine;
1767|                         ULONG
    | DuplicateLength = 0;
1768|
1769|                         // Copy
    | everything up to but excluding the second quote... (so
    | we can insert description)
1770|                         while ( pIn
    | < SecondQuote ) {
1771|                             *pOut++
    | = *pIn++;
1772|                         }

```

```

1773|
1774|                // insert
    | the switch description
1775|                while (
    | *pDesc ) {
1776|                    *pOut++
    | = *pDesc++;
1777|                }
1778|
1779|                *pOut++ =
    | *pIn++;    // copy the second quote
1780|
1781|                // insert
    | the switch itself
1782|                *pOut++ = '
    | ';
1783|
1784|                while (
    | *pSwitch ) {
1785|                    *pOut++
    | = *pSwitch++;
1786|                }
1787|
1788|                *pOut++ = '
    | ';
1789|
1790|                // copy the
    | rest of the original line...
1791|                while (
    | *pIn ) {
1792|                    *pOut++
    | = *pIn++;
1793|                }
1794|
1795|                *pOut =
    | '\0';
1796|
1797|    | DuplicateLength = strlen(DuplicateLine);
1798|
1799|                if (
    | *TempBytes + DuplicateLength >= TempBufferSize ) {
1800|                    Success
    | = FALSE;
1801|                    goto
    | BailOut;
1802|                }
1803|
1804|                strcpy (
    | &TempBuffer[*TempBytes], DuplicateLine );

```

```

1805|                                     *TempBytes
    | += DuplicateLength;
1806|                                     } // for
    | PsmSwitches
1807|                                     } // if
    | SecondQuote
1808|                                     } // if FirstQuote
1809|                                     } // if EqualSign
1810|                                     } // if
    | InOperatingSystemsSection
1811|                                     } // if ShouldCopyThisLine
1812|
1813|         OriginalIndex += LineLength;
1814|     } // while OriginalIndex <
    | OriginalBytesRead
1815|     } else {
1816|         WRITE_ERROR(L"!!! Internal Error:
    | OriginalBuffer==NULL in UpdateBootIniBuffer\n");
1817|     }
1818|     } else {
1819|         WRITE_ERROR(L"!!! Internal Error:
    | TempBuffer==NULL in UpdateBootIniBuffer\n");
1820|     }
1821|     } else {
1822|         WRITE_ERROR(L"!!! Internal Error:
    | TempBytes==NULL in UpdateBootIniBuffer\n");
1823|     }
1824|
1825| BailOut:
1826|     DEBUG_WRITE(L">>> UpdateBootIniBuffer returning
    | %s\n", (Success?L"TRUE":L"FALSE"));
1827|     return Success;
1828| }
1829|
1830| //-----
    | -----
    | --
1831|
1832|
1833| /*--- end of file bootini.c ---*/
1834|
1835|
1836|
1837| File Listing: File: CDP.h
1838|
1839| #define VER_COMPANYNAME_STR    "Columbia Data
    | Products, Inc."
1840| #define VER_LEGALTRADEMARKS_STR "PSM256 is a
    | trademark of " VER_COMPANYNAME_STR
1841|

```

```

1842| #define VER_LEGALCOPYRIGHT_YEARS "1995-2001"
1843| #define VER_LEGALCOPYRIGHT_STR  "Copyright \251 "
    | VER_LEGALCOPYRIGHT_YEARS " " VER_COMPANYNAME_STR
1844|
1845| #if DBG
1846| #define VER_DEBUG          VS_FF_DEBUG
1847| #else
1848| #define VER_DEBUG          0
1849| #endif
1850|
1851| #if BETA
1852| #define VER_PRODUCTBETA_STR  "BETA"
1853| #define VER_PRERELEASE      VS_FF_PRERELEASE
1854| #else
1855| #define VER_PRERELEASE      0
1856| #define VER_PRODUCTBETA_STR  ""
1857| #endif
1858|
1859| #define VER_FILEFLAGSMASK    VS_FFI_FILEFLAGSMASK
1860| #define VER_FILEOS          VOS_NT_WINDOWS32
1861| #define VER_FILEFLAGS      (VER_PRERELEASE |
    | VER_DEBUG)
1862|
1863|
1864|
1865| File Listing: File: precomp.h
1866|
1867| #ifndef __PSM_SS_PRECOMP
1868| #define __PSM_SS_PRECOMP 1
1869|
1870| #include <stdio.h>
1871| #include <stdlib.h>
1872| #include <stdarg.h>
1873| #include <string.h>
1874| #include <time.h>
1875| #include <process.h>
1876| #include <io.h>
1877| #include <errno.h>
1878| #include <conio.h>
1879| #include <fcntl.h>
1880| #include <windows.h>
1881| #include <windowsx.h>
1882| #include <commctrl.h> // includes the common control
    | header
1883| #include <tchar.h>
1884| #include <winioctl.h>
1885| #include <process.h>
1886| #include <direct.h>
1887|
1888| #include <winnt.h>

```

```

1889| #include <mountmgr.h>
1890|
1891| #include <psm.h>
1892| #include <psmlapi.h>
1893|
1894| #include <aclapi.h>
1895| #include <shellapi.h>
1896|
1897|
1898| #define getch    _getch
1899| #define wcsicmp  _wcsicmp
1900| #define strlwr   _strlwr
1901| #define wcsdup   _wcsdup
1902| #define stricmp  _stricmp
1903| #define strnicmp _strnicmp
1904|
1905| #include "psmoem.h"
1906| #include "ss.h"
1907| #include "vimage.h"
1908| #include "..\features\drbackup\drbackup.h"
1909|
1910| #endif /*__PSM_SS_PRECOMP*/
1911| /*--- end of file precomp.h ---*/
1912|
1913|
1914|
1915| File Listing: File: RESOURCE.h
1916|
1917| //{NO_DEPENDENCIES}
1918| // Microsoft Developer Studio generated include file.
1919| // Used by SBPSMAN.RC
1920| //
1921| #define VER_PRODUCTBUILD      3
1922| #define VER_PRODUCTVERSION_W  0x0101
1923|
1924| // Next default values for new objects
1925| //
1926| #ifdef APSTUDIO_INVOKED
1927| #ifndef APSTUDIO_READONLY_SYMBOLS
1928| #define _APS_NO_MFC            1
1929| #define _APS_NEXT_RESOURCE_VALUE 101
1930| #define _APS_NEXT_COMMAND_VALUE  40001
1931| #define _APS_NEXT_CONTROL_VALUE  1000
1932| #define _APS_NEXT_SYMED_VALUE   101
1933| #endif
1934| #endif
1935|
1936|
1937|
1938| File Listing: File: ss.c

```

```

1939|
1940| //define _UNICODE
1941|
1942| #include "precomp.h"
1943|
1944| #define PSM_CODE_CDP_TPSM      L"0KED9877DEAFEARS"
1945| #define PSM_CODE_CDP_TPSM_LICENSE  NULL
1946|
1947| #define FORWARD_CHRON  1
1948| #define REVERSE_CHRON  2
1949|
1950| // Map of priority as listed by webui
1951| #define PRIORITY_ALWAYS_KEEP      255
1952| #define PRIORITY_HIGHEST          254
1953| #define PRIORITY_HIGH              210
1954| #define PRIORITY_ABOVE_AVERAGE   165
1955| #define PRIORITY_NORMAL            125
1956| #define PRIORITY_BELOW_AVERAGE    85
1957| #define PRIORITY_LOW               40
1958| #define PRIORITY_LOWEST            0
1959|
1960| ULONG ViewSortOrder = REVERSE_CHRON;
1961| PWCHAR InVolumeMap[26]={0};
1962| WCHAR InVolumeMapData[26][100]={0};
1963| ULONG VolumeMapFlags[26];
1964| ULONG NumVolumes=0;
1965| WCHAR **OutVolumeMap=NULL;
1966| pSnapShot SnapShot;
1967| ULONG OutVolumeMapByteSize;
1968| tOpenTransactionInPersistentW In={0};
1969| tOpenTransactionOutW Out={0};
1970| ULONG GroupNumber=0;
1971| ULONG NumToKeep = -1;
1972| WCHAR PatternName[256]={0};
1973| BYTE Priority=PRIORITY_NORMAL;
1974| BYTE SnapShotFlags = PSM_SS_FLAG_P_READONLY;
1975| ULONG SaveTempOnExit = TRUE; // used only on
    | temporary snapshots (turn off with '-discardtemp').
1976| ULONG QuietMode = FALSE;
1977| ULONG QuietModeError = FALSE;
1978| ULONG DebugEnabled = FALSE;
1979|
1980| int DeleteExistingSnapShot( ULONG SnapShot );
1981|
1982| //-----
    | -----
1983| #if 0
1984| void PrintWin32Error( DWORD ErrorCode )
1985| {
1986|     LPVOID lpMsgBuf;

```

```

1987|
1988|   FormatMessage( FORMAT_MESSAGE_ALLOCATE_BUFFER |
    | FORMAT_MESSAGE_FROM_SYSTEM,
1989|               NULL, ErrorCode,
1990|               MAKELANGID(LANG_NEUTRAL,
    |   SUBLANG_DEFAULT),
1991|               (LPTSTR) &lpMsgBuf, 0, NULL );
1992|   WRITE_ERROR(L"%S\n", lpMsgBuf );
1993|   LocalFree( lpMsgBuf );
1994| }
1995| #else
1996| void PrintWin32Error( ULONG ErrorCode )
1997| {
1998|   WCHAR ErrorBuffer[1024];
1999|   ULONG Count;
2000|   HMODULE
    |   psmlapi=GetModuleHandle(TEXT("psmlapi.dll"));
2001|
2002|   SetLastError(0);
2003|   // see if NT system error messages or OTM error
2004|   Count = FormatMessageW(
2005|       FORMAT_MESSAGE_FROM_SYSTEM |
    |   FORMAT_MESSAGE_FROM_HMODULE,
2006|       psmlapi,
2007|       ErrorCode,
2008|       MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
2009|       ErrorBuffer,
2010|       sizeof(ErrorBuffer),
2011|       NULL);
2012|
2013| #if 0
2014|   // remove line feeds...
2015|   for(ULONG i=0;i<Count;i++) {
2016|       if((ErrorBuffer[i]==L'\n') ||
    |   (ErrorBuffer[i]==L'\r') ) {
2017|           ErrorBuffer[i] = L' ';
2018|       }
2019|   }
2020| #endif
2021|
2022|   if(Count!=0) {
2023|       WRITE_ERROR(L"%s\n",ErrorBuffer);
2024|   } else
2025|       WRITE_ERROR(L"Unknown error code %08x
    |   (%08x)\n",ErrorCode,GetLastError());
2026| }
2027| #endif
2028|
2029| HANDLE AbortEvent=INVALID_HANDLE_VALUE;
2030| WCHAR AbortMessage[256];

```

```

2031|
2032| // when the user presses Ctrl C, set the exit flag, and
    | return.
2033| // All the threads will exit, and the main thread will
    | be signaled to wake
2034| // since its waiting on the threads to finish.
2035| BOOL CtrlHandlerRoutine( DWORD dwCtrlType )
2036| {
2037|     switch(dwCtrlType) {
2038|         case CTRL_C_EVENT :
2039|             WRITE(L"Control-C pressed");
2040|             break;
2041|         case CTRL_BREAK_EVENT :
2042|             WRITE(L"Control-Break");
2043|             break;
2044|         case CTRL_CLOSE_EVENT :
2045|             WRITE(L"Close occuring");
2046|             break;
2047|         case CTRL_LOGOFF_EVENT :
2048|             WRITE(L"Logoff occurring");
2049|             break;
2050|         case CTRL_SHUTDOWN_EVENT :
2051|             WRITE(L"Shutdown occurring");
2052|             break;
2053|         default:
2054|             WRITE(L"Unknown Event occurred");
2055|     }
2056|     if(AbortEvent!=INVALID_HANDLE_VALUE) {
2057|         WRITE(AbortMessage);
2058|         SetEvent(AbortEvent);
2059|     } else {
2060|         WRITE(L".\n");
2061|     }
2062|     return TRUE;
2063| }
2064|
2065|
2066| PSECURITY_DESCRIPTOR GetSecuritySD(
2067|     BOOLEAN IncludeEveryone,
2068|     ULONG SystemRights,
2069|     ULONG AdminRights,
2070|     ULONG EveryoneRights
2071| )
2072| {
2073|     DWORD dwRes;
2074|     ULONG Err;
2075|     PSID pEveryoneSID = NULL, pAdminSID = NULL,
    | pSystemSID = NULL;
2076|     PACL pACL = NULL;
2077|     PSECURITY_DESCRIPTOR pSD = NULL;

```



```

2078| PSECURITY_DESCRIPTOR pSDRel = NULL;
2079| EXPLICIT_ACCESS ea[3];
2080| SID_IDENTIFIER_AUTHORITY SIDAuthWorld =
    | SECURITY_WORLD_SID_AUTHORITY;
2081| SID_IDENTIFIER_AUTHORITY SIDAuthLocal =
    | SECURITY_LOCAL_SID_AUTHORITY;
2082| SID_IDENTIFIER_AUTHORITY SIDAuthNT =
    | SECURITY_NT_AUTHORITY;
2083| ULONG Count;
2084|
2085|
2086| // Create a SID for the BUILTIN\Administrators
    | group.
2087|
2088| if(! AllocateAndInitializeSid( &SIDAuthNT, 2,
2089|     SECURITY_BUILTIN_DOMAIN_RID,
2090|     DOMAIN_ALIAS_RID_ADMINS,
2091|     0, 0, 0, 0, 0, 0,
2092|     &pAdminSID) ) {
2093|     printf( "AllocateAndInitializeSid Error %u\n",
    | GetLastError() );
2094|     goto Cleanup;
2095| }
2096|
2097| // Create a well-known SID for the Everyone group.
2098|
2099| if(! AllocateAndInitializeSid( &SIDAuthWorld, 1,
2100|     SECURITY_WORLD_RID,
2101|     0, 0, 0, 0, 0, 0, 0,
2102|     &pEveryoneSID) ) {
2103|     printf( "AllocateAndInitializeSid Error %u\n",
    | GetLastError() );
2104|     goto Cleanup;
2105| }
2106|
2107| // Create a well-known SID for the system group.
2108|
2109| if(! AllocateAndInitializeSid( &SIDAuthNT, 1,
2110|     SECURITY_LOCAL_SYSTEM_RID,
2111|     0, 0, 0, 0, 0, 0, 0,
2112|     &pSystemSID) ) {
2113|     printf( "AllocateAndInitializeSid Error %u\n",
    | GetLastError() );
2114|     goto Cleanup;
2115| }
2116|
2117| // Initialize an EXPLICIT_ACCESS structure for an
    | ACE.
2118| // The ACE will allow the Administrators group full
    | access to the key.

```

```

2119|
2120| ZeroMemory(&ea, 3 * sizeof(EXPLICIT_ACCESS));
2121|
2122| // system user
2123| ea[0].grfAccessPermissions = SystemRights;
2124| ea[0].grfAccessMode = SET_ACCESS;
2125| ea[0].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
2126| ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
2127| ea[0].Trustee.TrusteeType = TRUSTEE_IS_USER;
2128| ea[0].Trustee.ptstrName = (LPTSTR) pSystemSID;
2129|
2130| // admin
2131| ea[1].grfAccessPermissions = AdminRights;
2132| ea[1].grfAccessMode = SET_ACCESS;
2133| ea[1].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
2134| ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
2135| ea[1].Trustee.TrusteeType = TRUSTEE_IS_GROUP;
2136| ea[1].Trustee.ptstrName = (LPTSTR) pAdminSID;
2137|
2138| // everyone
2139| ea[2].grfAccessPermissions = EveryoneRights;
2140| ea[2].grfAccessMode = SET_ACCESS;
2141| ea[2].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
2142| ea[2].Trustee.TrusteeForm = TRUSTEE_IS_SID;
2143| ea[2].Trustee.TrusteeType =
    | TRUSTEE_IS_WELL_KNOWN_GROUP;
2144| ea[2].Trustee.ptstrName = (LPTSTR) pEveryoneSID;
2145|
2146|
2147| // Create a new ACL that contains the new ACEs.
2148|
2149| Count=2;
2150| if(IncludeEveryone) {
2151|     Count++;
2152| }
2153|
2154| dwRes = SetEntriesInAcl(Count, ea, NULL, &pACL);
2155| if (ERROR_SUCCESS != dwRes) {
2156|     SetLastError(dwRes);
2157|     printf( "SetEntriesInAcl Error %u\n",
    | GetLastError() );
2158|     goto Cleanup;
2159| }
2160|
2161| // Initialize a security descriptor.
2162|
2163| pSD = (PSECURITY_DESCRIPTOR) LocalAlloc(LPTR,

```

```

    | SECURITY_DESCRIPTOR_MIN_LENGTH);
2164|   if (pSD == NULL) {
2165|       printf( "LocalAlloc Error %u\n", GetLastError()
    | );
2166|       goto Cleanup;
2167|   }
2168|
2169|   if (!InitializeSecurityDescriptor(pSD,
    | SECURITY_DESCRIPTOR_REVISION)) {
2170|       printf( "InitializeSecurityDescriptor Error
    | %u\n",
2171|               GetLastError() );
2172|       goto Cleanup;
2173|   }
2174|
2175|   // Add the ACL to the security descriptor.
2176|
2177|   if (!SetSecurityDescriptorDacl(pSD,
2178|       TRUE,    // fDaclPresent flag
2179|       pACL,
2180|       FALSE)) // not a default DACL
2181|   {
2182|       printf( "SetSecurityDescriptorDacl Error %u\n",
    | GetLastError() );
2183|       goto Cleanup;
2184|   }
2185|
2186|   Count = 1024;
2187|   pSDRel = LocalAlloc(LPTR,Count);
2188|   if(!pSDRel) {
2189|       goto Cleanup;
2190|   }
2191|
2192|   if(MakeSelfRelativeSD(pSD,pSDRel,&Count)==0) {
2193|       LocalFree(pSDRel);
2194|       pSDRel=NULL;
2195|       goto Cleanup;
2196|   }
2197|
2198|   SetLastError(0);
2199| Cleanup:
2200|
2201|   Err = GetLastError();
2202|
2203|   if (pEveryoneSID) {
2204|       FreeSid(pEveryoneSID);
2205|   }
2206|   if (pSystemSID) {
2207|       FreeSid(pSystemSID);
2208|   }

```

```

2209|  if (pAdminSID) {
2210|      FreeSid(pAdminSID);
2211|  }
2212|  if (pACL) {
2213|      LocalFree(pACL);
2214|  }
2215|
2216|  if (pSD) {
2217|      LocalFree(pSD);
2218|  }
2219|  SetLastError(Err);
2220|  return pSDRel;
2221| }
2222|
2223| //-----
2224| | -----
2225| ULONG GetPsmRegistryDword (
2226|     const WCHAR * ValueName,
2227|     ULONG      DefaultValue )
2228| {
2229|     ULONG Err=0;
2230|     HKEY Key = INVALID_HANDLE_VALUE;
2231|     const WCHAR *RegPath =
2232|         | L"SYSTEM\\CurrentControlSet\\Services\\PSMan5\\Persisten
2233|         | t\\";
2234|     ULONG Value = DefaultValue;
2235|
2236|     Err = RegOpenKeyExW (
2237|         HKEY_LOCAL_MACHINE,
2238|         RegPath,
2239|         0,
2240|         KEY_READ,
2241|         &Key );
2242|
2243|     if ( Err == 0 ) {
2244|         ULONG TempValue = 0;
2245|         ULONG DataSize = sizeof(TempValue);
2246|         Err = RegQueryValueExW (
2247|             Key,
2248|             ValueName,
2249|             NULL, // reserved
2250|             NULL, // address of buffer for value type
2251|             (char*)&TempValue,
2252|             &DataSize );
2253|
2254|         if ( Err == 0 ) {
2255|             Value = TempValue;
2256|         }
2257|     }

```

```

2256|     RegCloseKey(Key);
2257|     Key = INVALID_HANDLE_VALUE;
2258| } else {
2259|     DEBUG_WRITE(L">>> GetPsmRegistryDword: error
    | %08x opening '%s'\n",RegPath);
2260| }
2261|
2262| return Value;
2263| }
2264|
2265| //-----
    | -----
2266| // GetSnapShotCreationFlags - This function
    | translates PSM_SS_FLAG values into
2267| // PSM_FLAG values. In other words, it translates a
    | snapshot's flag byte into
2268| // a snapshot creation flag dword.
2269|
2270| ULONG GetSnapShotCreationFlags ( BYTE
    | UserSnapShotFlags, ULONG *Flags )
2271| {
2272|     ULONG Err = 0;
2273|     if ( Flags != NULL ) {
2274|         ULONG EnableForcedSnapShot =
            | GetPsmRegistryDword (L"EnableForcedSnapShot", 1);
2275|
2276|         *Flags = PSM_FLAG_PAUSE_ON_IO;
2277|         if ( EnableForcedSnapShot ) {
2278|             *Flags |= PSM_FLAG_FORCE_SNAPSHOT;
2279|             DEBUG_WRITE(L">>> GetSnapShotCreationFlags:
                | Snapshot will be forced if quiescence times out.\n");
2280|         } else {
2281|             DEBUG_WRITE(L">>> GetSnapShotCreationFlags:
                | Snapshot will be aborted if quiescence times out.\n");
2282|         }
2283|
2284|         if ( PSM_SS_IsPersistent(UserSnapShotFlags) ) {
2285|             *Flags |= PSM_FLAG_PERSISTENT_SNAPSHOT;
2286|             DEBUG_WRITE(L">>> GetSnapShotCreationFlags:
                | Snapshot will be persistent.\n");
2287|         } else {
2288|             DEBUG_WRITE(L">>> GetSnapShotCreationFlags:
                | Snapshot will be temporary.\n");
2289|             if ( SaveTempOnExit ) {
2290|                 *Flags |= PSM_FLAG_SAVE_TEMP_ON_EXIT;
2291|                 DEBUG_WRITE(L">>>
                    | GetSnapShotCreationFlags: Temporary snapshot will be
                    | retained after thread exits.\n");
2292|             } else {
2293|                 DEBUG_WRITE(L">>>

```

```

    | GetSnapShotCreationFlags: Temporary snapshot will be
    | discarded after thread exits.\n");
2294|     }
2295| }
2296| } else {
2297|     Err = ERROR_INVALID_PARAMETER;
2298| }
2299|
2300|     DEBUG_WRITE(L">>> GetSnapShotCreationFlags()
    | returning Err=%08x,
    | Flags=%08x\n",Err,(Flags?(*Flags):0));
2301|     return Err;
2302| }
2303|
2304| //-----
    | -----
2305|
2306| int MakeNewSnapShot()
2307| {
2308|     ULONG Err=0;
2309|
2310|     UpdateBootIni();
2311|
2312|
    | memset(&In,0,sizeof(tOpenTransactionInPersistentW));
2313|
2314|     In.Size = sizeof(tOpenTransactionInPersistentW);
2315|     Err = GetSnapShotCreationFlags(SnapShotFlags,
    | &In.Flags);
2316|
2317|     if ( Err == 0 ) {
2318|         In.CallerPrivateUse = (PVOID)GroupNumber;
2319|         In.NumToKeep      = NumToKeep;
2320|
2321|         // this is the snapshot name or L"" if none
2322|         wcscpy(In.SnapShotName,PatternName);
2323|
2324|         // not used
2325|         In.ErrorEvent      = NULL;
2326|
2327|         // set event so they can cancel waiting for q
    | period
2328|         AbortEvent = In.AbortEvent = CreateEvent(
    | NULL, TRUE, FALSE, NULL );
2329|
2330|         // Set our Control - C handler
2331|         wcscpy(AbortMessage,L", cancelling creation of
    | snapshot.\n");
2332|         SetConsoleCtrlHandler(
    | (PHANDLER_ROUTINE)CtrlHandlerRoutine, TRUE );

```

```

2333|
2334|     In.Priority = Priority;
2335|     In.SnapShotFlags = SnapShotFlags;
2336|
2337|     OutVolumeMapByteSize = 32768;
2338|     OutVolumeMap = malloc(OutVolumeMapByteSize);
2339|     if(OutVolumeMap) {
2340|
2341|         Err = Psm_CreateSnapShotW(
2342|             (pOpenTransactionInW)&In,
2343|             NumVolumes,
2344|             InVolumeMap,
2345|             VolumeMapFlags,
2346|             OutVolumeMapByteSize,
2347|             OutVolumeMap,
2348|             &Out,
2349|             &SnapShot,
2350|             NULL,
2351|             NULL );
2352|
2353|         if(!Err) {
2354|             ULONG i;
2355|             WRITE(L"Snapshot created
| successfully\n");
2356|             for(i=0;i<NumVolumes;i++) {
2357|                 WRITE(L""s' =
| '%s\n",InVolumeMap[i],OutVolumeMap[i]);
2358|             }
2359|         } else {
2360|             WRITE_ERROR(L"Error %08x creating
| snapshot\n",Err);
2361|         }
2362|
2363|         free(OutVolumeMap);
2364|         OutVolumeMap = NULL;
2365|     } else {
2366|         WRITE_ERROR(L"Out of memory\n");
2367|         Err = ERROR_OUTOFMEMORY;
2368|     }
2369| }
2370|
2371| if(AbortEvent) {
2372|     CloseHandle(AbortEvent);
2373|     AbortEvent = INVALID_HANDLE_VALUE;
2374| }
2375|
2376| if(Err) {
2377|     PrintWin32Error(Err);
2378| }
2379|

```

```

2380|    return Err;
2381| }
2382|
2383| //-----
    | -----
2384|
2385| #include "vimage.inc"
2386|
2387| //-----
    | -----
2388|
2389| int IsSnapShotAlwaysKeep( ULONG SnapShot )
2390| {
2391|     ULONG Err=0;
2392|     tPSM_GetKernelSnapShotInfoW Info;
2393|
2394|     if(SnapShot) {
2395|         Err =
            | Psm_GetKernelSnapShotInfoW((PVOID)SnapShot,&Info);
2396|         if(!Err) {
2397|             if(Info.Priority == PRIORITY_ALWAYS_KEEP) {
2398|                 return TRUE;
2399|             }
2400|         } else {
2401|         }
2402|     }
2403|     return FALSE;
2404| }
2405|
2406| ULONG AlwaysDelete = FALSE;
2407|
2408| int OkayToDelete( ULONG SnapShot )
2409| {
2410|     WCHAR ch;
2411|
2412|     if(IsSnapShotAlwaysKeep(SnapShot)) {
2413|         if(!AlwaysDelete) {
2414|             tPSM_GetKernelSnapShotInfoW Info={0};
2415|             FILETIME NewFileTime={0};
2416|             SYSTEMTIME SystemTime={0};
2417|
2418|             | Psm_GetKernelSnapShotInfoW((PVOID)SnapShot,&Info);
2419|
                | FileTimeToLocalFileTime((FILETIME*)&Info.SnapShotTime,&N
                | ewFileTime);
2420|
                | FileTimeToSystemTime(&NewFileTime,&SystemTime);
2421|
2422|             WRITE(L""%08x | %2d/%02d/%4d

```



```

    | %2d:%02d:%02d.%03d | %s\n  is set to always keep,
    | delete anyway (Y/N/A)? ",
2423|         SnapShot,
2424|         SystemTime.wMonth,
2425|         SystemTime.wDay,
2426|         SystemTime.wYear,
2427|         SystemTime.wHour,
2428|         SystemTime.wMinute,
2429|         SystemTime.wSecond,
2430|         SystemTime.wMilliseconds,
2431|         Info.Directory
2432|     );
2433|     do {
2434|         ch = (WCHAR) toupper(getch());
2435|     } while((ch!='Y') && (ch!='N') &&
    | (ch!='A'));
2436|
2437|     WRITE(L"%c\n",ch);
2438|
2439|     if(ch=='N') {
2440|         return FALSE;
2441|     } else if(ch=='A') {
2442|         AlwaysDelete = TRUE;
2443|     }
2444| }
2445| }
2446| return TRUE;
2447| }
2448|
2449| int DeleteExistingSnapShot( ULONG SnapShot )
2450| {
2451|     ULONG Err=0;
2452|     BOOLEAN GotData = FALSE;
2453|     tPSM_GetKernelSnapShotInfo Info;
2454|     SYSTEMTIME SystemTime;
2455|
2456|     if(SnapShot) {
2457|         Err =
            | Psm_GetKernelSnapShotInfo((PVOID)SnapShot,&Info);
2458|         if(!Err) {
2459|             FILETIME NewFileTime;
2460|
            | FileTimeToLocalFileTime((FILETIME*)&Info.SnapShotTime,&N
            | ewFileTime);
2461|
            | FileTimeToSystemTime(&NewFileTime,&SystemTime);
2462|             GotData=TRUE;
2463|         }
2464|
2465|         if(OkayToDelete(SnapShot)) {

```

```

2466|         Err = Psm_DestroySnapShot((PVOID)SnapShot);
2467|         if(!Err) {
2468|             if(!GotData) {
2469|                 WRITE(L"Success deleting snapshot
| %08x\n",SnapShot);
2470|             } else {
2471|                 WRITE(L"%08x | %2d/%02d/%4d
| %2d:%02d:%02d.%03d successfully deleted\n",
2472|                     SnapShot,
2473|                     SystemTime.wMonth,
2474|                     SystemTime.wDay,
2475|                     SystemTime.wYear,
2476|                     SystemTime.wHour,
2477|                     SystemTime.wMinute,
2478|                     SystemTime.wSecond,
2479|                     SystemTime.wMilliseconds );
2480|             }
2481|
2482|         } else {
2483|             WRITE_ERROR(L"Error %08x deleting
| snapshot %08x\n",Err,SnapShot);
2484|         }
2485|     }
2486| }
2487| if(Err) {
2488|     PrintWin32Error(Err);
2489| }
2490| return 0;
2491| }
2492|
2493| //-----
| -----
2494|
2495| PSMSTATUS PSMAPI Psm_GetPersistentSnapShots( PVOID
| Buffer, ULONG BufferSize );
2496| typedef struct sPSM_GetPersistentSnapShotsOut {
2497|     PVOID KernelPointers[MAX_NUMBER_OF_SNAPSHOTS];
2498| } tPSM_GetPersistentSnapShotsOut,
| *pPSM_GetPersistentSnapShotsOut;
2499|
2500| #define VIEW_NORMAL 0
2501| #define VIEW_SHORT 1
2502| #define VIEW_LONG 2
2503|
2504| WCHAR *GetSSFlags( BYTE Flags )
2505| {
2506|     switch (Flags & PSM_SS_FLAG_TYPE_MASK) {
2507|         case PSM_SS_FLAG_P_READWRITE : return
| L"R/W ";
2508|         case PSM_SS_FLAG_P_READWRITE_PVW : return

```

```

    | L"R/W P";
2509|     case PSM_SS_FLAG_P_READONLY      : return
    | L"R/O ";
2510|     case PSM_SS_FLAG_T_READONLY      : return
    | L"T R/O";
2511|     case PSM_SS_FLAG_T_READWRITE     : return
    | L"T R/W";
2512|     default:
2513|         return L"Unk ";
2514|     }
2515| }
2516|
2517|
2518| void SortInChronologicalOrder (
2519|     tPSM_GetPersistentSnapShotsOut *ssArray,
2520|     ULONG sortOrder )
2521| {
2522|     // Figure out how many snapshots there are.
2523|     // While doing that, find out when each was
    | created.
2524|
2525|     __int64 ssTime [MAX_NUMBER_OF_SNAPSHOTS];
2526|     unsigned numSnapShots = MAX_NUMBER_OF_SNAPSHOTS;
2527|     tPSM_GetKernelSnapShotInfoW ssInfo;
2528|     ULONG status=0;
2529|     unsigned i, j;
2530|     for ( i=0; i<numSnapShots; ++i ) {
2531|         if ( ssArray->KernelPointers[i] == NULL ) {
2532|             numSnapShots = i;
2533|             break;
2534|         }
2535|
2536|         status =
            | Psm_GetKernelSnapShotInfoW(ssArray->KernelPointers[i],&s
            | sInfo);
2537|         if ( status == STATUS_SUCCESS ) {
2538|             ssTime[i] = ssInfo.SnapShotTime.QuadPart;
2539|         } else {
2540|             ssTime[i] = 0;
2541|         }
2542|     }
2543|
2544|     if ( numSnapShots > 1 ) {
2545|
2546|         // Now we have parallel arrays of times and
            | snapshot pointers.
2547|         // Sort them in tandem.
2548|
2549|         for ( i=0; i<numSnapShots-1; ++i ) {
2550|             // On each loop of 'i', find the thing that

```

```

    | goes to index i.
2551|         // Search index i+1..numSnapShots-1 for the
    | largest time left.
2552|         unsigned best = i;
2553|         for ( j=i+1; j<numSnapShots; ++j ) {
2554|             BOOLEAN better =
    | (sortOrder==REVERSE_CHRON) ?
2555|             (ssTime[j] > ssTime[best]) :
2556|             (ssTime[j] < ssTime[best]);
2557|
2558|             if ( better ) {
2559|                 best = j;
2560|             }
2561|         }
2562|
2563|         if ( best != i ) {
2564|             // Swap what is at i with the best
    | thing found after it
2565|             PVOID swapPointer =
    | ssArray->KernelPointers[i];
2566|             __int64 swapTime = ssTime[i];
2567|
2568|             ssTime[i] = ssTime[best];
2569|             ssTime[best] = swapTime;
2570|
2571|             ssArray->KernelPointers[i] =
    | ssArray->KernelPointers[best];
2572|             ssArray->KernelPointers[best] =
    | swapPointer;
2573|         }
2574|     }
2575| }
2576| }
2577|
2578|
2579| int ListExistingSnapShots( ULONG Type )
2580| {
2581|     tPSM_GetPersistentSnapShotsOut *Out = 0;
2582|     tPSM_GetKernelSnapShotInfoW Info;
2583|     ULONG Err = 0;
2584|     SYSTEMTIME SystemTime;
2585|
2586|     Out =
    | malloc(sizeof(tPSM_GetPersistentSnapShotsOut));
2587|     if(Out) {
2588|         Err =
    | Psm_GetPersistentSnapShots(Out,sizeof(tPSM_GetPersistent
    | SnapShotsOut));
2589|         if(!Err) {
2590|             ULONG FoundOne=0;

```

```

2591|         ULONG i;
2592|
2593|         SortInChronologicalOrder (Out,
    | ViewSortOrder);
2594|
2595|         for(i=0;i<MAX_NUMBER_OF_SNAPSHOTS;i++) {
2596|             if(Out->KernelPointers[i]==NULL) {
2597|                 // break at first null
2598|                 break;
2599|             }
2600|             if(!FoundOne) {
2601|                 // print header
2602|                 switch ( Type) {
2603|                     case VIEW_NORMAL:
2604|                         WRITE(L" Id |
    | Date/time of snapshots | Status | Directory\n");
2605|                         break;
2606|                     case VIEW_SHORT:
2607|                         WRITE(L"Date/time of
    | snapshots | Directory\n");
2608|                         break;
2609|                     case VIEW_LONG:
2610|                         WRITE(L" Id |
    | Date/time of snapshots | Unq | Group | Status |
    | Pr. | Flags | Directory\n");
2611|                         break;
2612|                 };
2613|             }
2614|             FoundOne = TRUE;
2615|             Err =
    | Psm_GetKernelSnapShotInfoW(Out->KernelPointers[i],&Info)
    | ;
2616|
2617|             if(!Err) {
2618|                 ULONG BufferSize=65536;
2619|                 WCHAR *Buffer = 0;
2620|
2621|                 // get the list of volumes for this
    | snapshot
2622|                 do {
2623|                     Buffer = (WCHAR
    | *)malloc(BufferSize);
2624|                     if(Buffer) {
2625|                         Err =
    | Psm_GetKernelSnapShotVolumesW(
2626|                             Out->KernelPointers[i],
2627|                             Buffer,
2628|                             BufferSize);
2629|
2630|                         if(Err==ERROR_MORE_DATA) {

```

```

2631|                Err = 0;
2632|                BufferSize *= 2;
2633|                free(Buffer);
2634|                Buffer = NULL;
2635|            } else {
2636|                if(Err) {
2637|                    WRITE_ERROR(L"%08x
| | Error %08x getting volume
| list\n",Out->KernelPointers[i],Err);
2638|                }
2639|            }
2640|        } else {
2641|            Err = ERROR_OUTOFMEMORY;
2642|        }
2643|    } while(Err==ERROR_MORE_DATA);
2644|
2645|    if(!Err && Buffer) {
2646|        WCHAR *p = 0;
2647|        FILETIME NewFileTime;
2648|
| FileTimeToLocalFileTime((FILETIME*)&Info.SnapShotTime,&N
| ewFileTime);
2649|
| FileTimeToSystemTime(&NewFileTime,&SystemTime);
2650|
2651|        p = Buffer;
2652|        while(*p) {
2653|            switch ( Type ) {
2654|                case VIEW_NORMAL:
2655|                    WRITE(L"%08x |
| %2d/%02d/%4d %2d:%02d:%02d.%03d | %08x | %s\\%s\n",
2656|
| Out->KernelPointers[i],
2657|
| SystemTime.wMonth,
2658|
| SystemTime.wDay,
2659|
| SystemTime.wYear,
2660|
| SystemTime.wHour,
2661|
| SystemTime.wMinute,
2662|
| SystemTime.wSecond,
2663|
| SystemTime.wMilliseconds,
2664|                    Info.Status,
2665|                    p,
2666|                    Info.Directory

```

```

| );
2667|             break;
2668|
2669|             case VIEW_SHORT:
2670|
2671|                 | WRITE(L"%2d/%02d/%4d %2d:%02d:%02d.%03d | %s\\%s\n",
2672|                 | SystemTime.wMonth,
2673|                 | SystemTime.wDay,
2674|                 | SystemTime.wYear,
2675|                 | SystemTime.wHour,
2676|                 | SystemTime.wMinute,
2677|                 | SystemTime.wSecond,
2678|                 | SystemTime.wMilliseconds,
2679|                 p,
2680|                 Info.Directory
2681|             | );
2682|             break;
2683|
2684|             case VIEW_LONG: {
2685|                 unsigned char
2686|                 | priority=Info.Priority;
2687|                 | WRITE(L"%08x |
2688|                 | %2d/%02d/%4d %2d:%02d:%02d.%03d | %3u | %08x | %08x |
2689|                 | %3u | %-5.5s | %s\\%s\n",
2690|                 | Out->KernelPointers[i],
2691|                 | SystemTime.wMonth,
2692|                 | SystemTime.wDay,
2693|                 | SystemTime.wYear,
2694|                 | SystemTime.wHour,
2695|                 | SystemTime.wMinute,
2696|                 | SystemTime.wSecond,
2697|                 | SystemTime.wMilliseconds,
2698|                 Info.Instance,
2699|                 | Info.CallerPrivateUse,

```

```

2695|             Info.Status,
2696|             priority,
2697|
2698|             | GetSSFlags(Info.SnapShotFlags),
2699|             p,
2700|             Info.Directory
2701|         | );
2702|         break;
2703|     }
2704| } // switch
2705| DEBUG_WRITE(L">>>
2706| | SnapShotTime = %016l64x\n",Info.SnapShotTime);
2707|         p += wcslen(p) + 1;
2708|     } // while (*p)
2709|     free(Buffer);
2710| } // if !Err
2711| } else {
2712|     WRITE(L"%08x | No info available
2713| | %08x\n",Out->KernelPointers[i],Err);
2714| }
2715| }
2716| if(!FoundOne) {
2717|     WRITE_ERROR(L"No persistent snapshots
2718| | exist\n");
2719| }
2720| } else {
2721|     WRITE_ERROR(L"Error %08x getting snapshot
2722| | list\n",Err);
2723| }
2724| }
2725| free (Out);
2726| Out = NULL;
2727| } else {
2728|     WRITE_ERROR(L"Out of memory\n");
2729| }
2730| if(Err) {
2731|     PrintWin32Error(Err);
2732| }
2733| return 0;
2734| }
2735|
2736| //-----
2737| | -----
2738|
2739| int DeleteAllExistingSnapShots()
2740| {
2741|     tPSM_GetPersistentSnapShotsOut *Out = 0;
2742|     ULONG Err=0, NumDeleted=0;
2743|

```



```

2738| Out =
    | malloc(sizeof(tPSM_GetPersistentSnapShotsOut));
2739| if(Out) {
2740|     Err =
        | Psm_GetPersistentSnapShots(Out,sizeof(tPSM_GetPersistent
        | SnapShotsOut));
2741|     if(!Err) {
2742|         BOOLEAN FoundOne = FALSE;
2743|         ULONG i;
2744|         for(i=0;i<MAX_NUMBER_OF_SNAPSHOTS;i++) {
2745|             if(Out->KernelPointers[i]==NULL) {
2746|                 break;
2747|             }
2748|             FoundOne = TRUE;
2749|             Err = DeleteExistingSnapShot ((ULONG)
                | (Out->KernelPointers[i]));
2750|             if ( Err == 0 ) {
2751|                 ++NumDeleted;
2752|             }
2753|         }
2754|         if(!FoundOne) {
2755|             WRITE_ERROR(L"No persistent snapshots
                | exist\n");
2756|         }
2757|     } else {
2758|         WRITE_ERROR(L"Error %08x getting snapshot
                | list\n",Err);
2759|     }
2760|
2761|     free (Out);
2762|     Out = NULL;
2763| } else {
2764|     WRITE_ERROR(L"Out of memory\n");
2765|     Err = ERROR_OUTOFMEMORY;
2766| }
2767|
2768| if(Err) {
2769|     PrintWin32Error(Err);
2770| }
2771| WRITE ( L"Deleted %lu snapshot%S.\n", NumDeleted,
    | (NumDeleted==1)?L"":L"s" );
2772| return Err;
2773| }
2774|
2775| int DeleteAllSnapShotsInGroup( ULONG Group )
2776| {
2777|     tPSM_GetPersistentSnapShotsOut *Out = 0;
2778|     tPSM_GetKernelSnapShotInfoW Info;
2779|     ULONG Err=0, NumDeleted=0;
2780|

```

```

2781| Out =
    | malloc(sizeof(tPSM_GetPersistentSnapShotsOut));
2782| if(Out) {
2783|     Err =
        | Psm_GetPersistentSnapShots(Out,sizeof(tPSM_GetPersistent
        | SnapShotsOut));
2784|     if(!Err) {
2785|         BOOLEAN FoundOne = FALSE;
2786|         ULONG i;
2787|         for(i=0;i<MAX_NUMBER_OF_SNAPSHOTS;i++) {
2788|             if(Out->KernelPointers[i]==NULL) {
2789|                 break;
2790|             }
2791|
2792|             if((Err =
                | Psm_GetKernelSnapShotInfoW(Out->KernelPointers[i],&Info)
                | )==0) {
2793|                 if(Info.CallerPrivateUse ==
                    | (PVOID)Group ) {
2794|                     FoundOne = TRUE;
2795|                     Err = DeleteExistingSnapShot
                        | ((ULONG) (Out->KernelPointers[i]));
2796|                     if ( Err == 0 ) {
2797|                         ++NumDeleted;
2798|                     }
2799|                 }
2800|             } else {
2801|                 WRITE_ERROR(L"Error %08x getting
                    | snapshot info for %08x\n",Err,Out->KernelPointers[i]);
2802|             }
2803|         }
2804|
2805|         if(!FoundOne) {
2806|             WRITE_ERROR(L"No persistent snapshots
                | exist\n");
2807|         }
2808|     } else {
2809|         WRITE_ERROR(L"Error %08x getting snapshot
                | list\n",Err);
2810|     }
2811|     free (Out);
2812|     Out = NULL;
2813| } else {
2814|     WRITE_ERROR(L"Out of memory\n");
2815|     Err = ERROR_OUTOFMEMORY;
2816| }
2817|
2818| if(Err) {
2819|     PrintWin32Error(Err);
2820| }

```

```

2821|   WRITE ( L"Deleted %lu snapshot%s for group
    | %08x.\n", NumDeleted, (NumDeleted==1)?L"":L"s" ,Group);
2822|   return Err;
2823| }
2824|
2825|
2826| //-----
    | -----
2827|
2828|
2829| int InstallPsm()
2830| {
2831|   CHAR rebootNeeded = 0;
2832|   PSMSTATUS status = Psm_InstallPsm ( &rebootNeeded
    | );
2833|   if ( status == 0 ) {
2834|       WRITE ( L"PSM has been installed.\n" );
2835|       if ( rebootNeeded ) {
2836|           WRITE ( L"You will need to reboot the
    | computer for changes to take effect.\n" );
2837|       }
2838|   } else if ( status == ERROR_SERVICE_EXISTS ) {
2839|       WRITE ( L"PSM was already installed.\n" );
2840|   } else {
2841|       WRITE_ERROR ( L"Error 0x%08lx occurred trying
    | to install PSM.\n", (unsigned long)status );
2842|       PrintWin32Error ( status );
2843|   }
2844|
2845|   return status;
2846| }
2847|
2848| //-----
    | -----
2849|
2850| int UninstallPsm()
2851| {
2852|   CHAR rebootNeeded = 0;
2853|   PSMSTATUS status = Psm_UnInstallPsm ( &rebootNeeded
    | );
2854|   if ( status == 0 ) {
2855|       WRITE ( L"PSM has been uninstalled.\n" );
2856|       if ( rebootNeeded ) {
2857|           WRITE ( L"You will need to reboot the
    | computer for changes to take effect.\n" );
2858|       }
2859|   } else {
2860|       WRITE_ERROR ( L"Error 0x%08lx occurred trying
    | to uninstall PSM.\n", (unsigned long)status );
2861|       PrintWin32Error ( status );

```

```

2862|    }
2863|
2864|    return status;
2865| }
2866|
2867| //-----
    | -----
2868|
2869| int TestList()
2870| {
2871|     // Test new functions for listing both persistent
    | and temporary snapshots.
2872|     // Also test functions for enumerating PSM'ed
    | volumes associated with a snapshot.
2873|
2874|     pSnapShot snapshotTable [1024];
2875|     const ULONG TABLE_SIZE = sizeof(snapshotTable) /
    | sizeof(snapshotTable[0]);
2876|     ULONG numSnapshots = 0;
2877|     int errorlevel = 0;
2878|
2879|     PSMSTATUS status = Psm_GetActiveSnapshotTable (
2880|         &numSnapshots,
2881|         snapshotTable,
2882|         TABLE_SIZE );
2883|
2884|     if ( status == 0 ) {
2885|         ULONG i=0;
2886|         ULONG iterator = 0;
2887|         WCHAR *volumeName = 0;
2888|
2889|         WRITE ( L"Found %lu active snapshot%S\n",
2890|             numSnapshots,
2891|             (numSnapshots==1) ? L"" : L"s" );
2892|
2893|         for ( i=0; i<numSnapshots && status==0; ++i ) {
2894|             int numVolumesFound = 0;
2895|
2896|             WRITE ( L"snapshot pointer = %p\n",
    | snapshotTable[i] );
2897|             status = Psm_FindFirstVolumeForSnapshot (
2898|                 snapshotTable[i],
2899|                 &volumeName,
2900|                 &iterator );
2901|
2902|             if ( status == 0 ) {
2903|                 if ( volumeName ) {
2904|                     WRITE ( L"    volume[%d] = %S\n",
    | numVolumesFound++, volumeName );
2905|                     while ( volumeName && status==0 ) {

```

```

2906|         status =
    | Psm_FindNextVolumeForSnapshot (
2907|             snapshotTable[i],
2908|             &volumeName,
2909|             &iterator );
2910|
2911|         if ( status == 0 ) {
2912|             if ( volumeName ) {
2913|                 WRITE ( L"
    | volume[%d] = %S\n", numVolumesFound++, volumeName );
2914|             }
2915|         } else {
2916|             WRITE_ERROR ( L"Error %08x
    | returned from Psm_FindNextVolumeForSnapshot()\n",
    | status );
2917|             errorlevel = 3;
2918|         }
2919|     }
2920| }
2921| } else {
2922|     WRITE_ERROR ( L"Error %08x returned
    | from Psm_FindFirstVolumeForSnapshot()\n", status );
2923|     errorlevel = 2;
2924| }
2925| }
2926| } else {
2927|     WRITE_ERROR ( L"Error %08x returned from
    | Psm_GetActiveSnapshotTable()\n", status );
2928|     errorlevel = 1;
2929| }
2930|
2931| return errorlevel;
2932| }
2933|
2934| //-----
    | -----
2935| ULONG GetDriveLetterFromWin32Name( WCHAR *Win32Name,
    | WCHAR *DriveLetter)
2936| {
2937|     WCHAR *Buffer,*p;
2938|     WCHAR Name[256];
2939|     BOOLEAN FoundOne=FALSE;
2940|     WCHAR LookingFor[256];
2941|
2942|     // its in \\.\?\\Volume{ we need Volume{
2943|     QueryDosDeviceW(Win32Name+4,LookingFor,256);
2944|
2945|     Buffer =
    | (WCHAR*)LocalAlloc(LPTR,65536*sizeof(WCHAR));
2946|     QueryDosDeviceW(NULL,Buffer,65536);

```



```

2991|             mbTotal,
2992|             mbMax,
2993|             InVolumeMap[i]
2994|         );
2995|     } else {
2996|         WRITE( L"
    | | %s\n",InVolumeMap[i]);
2997|     }
2998|     } else if ( AutoEnumerated &&
    | status==ERROR_NOT_FOUND ) {
2999|         DEBUG_WRITE(L">>> We thought volume
    | '%s' was PSMable, but cannot get its cache
    | info!\n",InVolumeMap[i]);
3000|         status = STATUS_SUCCESS;
3001|     } else {
3002|         WRITE_ERROR ( L"Error %08x retrieving
    | info    | %s\n", status,InVolumeMap[i] );
3003|     }
3004| }
3005| } else {
3006|     WCHAR Vol[256];
3007|     HANDLE VolHandle =
    | FindFirstVolumeW(Vol,256*sizeof(WCHAR));
3008|     if(VolHandle!=INVALID_HANDLE_VALUE) {
3009|         do {
3010|             if(Vol[wcslen(Vol)-1]==L'\\') {
3011|                 Vol[wcslen(Vol)-1] = L'0';
3012|             }
3013|
    | if(GetDriveLetterFromWin32Name(Vol,InVolumeMapData[NumVo
    | lumes])!=0) {
3014|         | wcscpy(InVolumeMapData[NumVolumes],Vol);
3015|     }
3016|
    | if(((InVolumeMapData[NumVolumes][0]==L'A') &&
    | (InVolumeMapData[NumVolumes][1]==L':')) ||
3017|     | (((InVolumeMapData[NumVolumes][0]==L'A') &&
    | (InVolumeMapData[NumVolumes][1]==L':')))) {
3018|         // skip floppies
3019|     } else {
3020|
    | if(Psm_CanBePSMedW(InVolumeMapData[NumVolumes])) {
3021|
    | InVolumeMap[NumVolumes]=InVolumeMapData[NumVolumes];
3022|         NumVolumes++;
3023|     }
3024| }
3025| }

```

```

    | while(FindNextVolumeW(VolHandle,Vol,256));
3026|     FindVolumeClose(VolHandle);
3027|     if(NumVolumes) {
3028|         AutoEnumerated = TRUE;
3029|         goto ListVols;
3030|     }
3031| }
3032| }
3033|
3034| if(status) {
3035|     PrintWin32Error(status);
3036| }
3037| return NT_SUCCESS(status) ? 0 : 1;
3038| }
3039|
3040| int CreateCache()
3041| {
3042|     PSMSTATUS status=0;
3043|     ULONG Err=0;
3044|
3045|     UpdateBootIni();
3046|
3047|     if(NumVolumes>0) {
3048|         // set event so they can cancel waiting for q
3049|         | period
3049|         AbortEvent = CreateEvent( NULL, TRUE, FALSE,
3050|         | NULL );
3050|
3051|         // Set our Control - C handler
3052|         wcscpy(AbortMessage,L", cancelling creating
3053|         | cache files.\n");
3053|         SetConsoleCtrlHandler(
3054|         | (PHANDLER_ROUTINE)CtrlHandlerRoutine, TRUE );
3054|
3055|         status = Psm_CreateFilesW( NumVolumes,
3056|         | InVolumeMap, AbortEvent );
3056|         if(status) {
3057|             PrintWin32Error(status);
3058|         }
3059|
3060|         if(AbortEvent) {
3061|             CloseHandle(AbortEvent);
3062|             AbortEvent = INVALID_HANDLE_VALUE;
3063|         }
3064|
3065|     } else {
3066|         WRITE_ERROR(L"No volumes specified\n");
3067|     }
3068|
3069|     return status;

```



```

3070| }
3071|
3072|
3073| //-----
    | -----
3074|
3075|
3076| int IsInstalled()
3077| {
3078|     tPSM_VersionInfo version = {0};
3079|     PSMSTATUS status = Psm_IsInstalled (
3080|         sizeof(version),
3081|         &version );
3082|
3083|     switch ( status ) {
3084|         case PSM_ERROR_REBOOT_NEEDED: WRITE_ERROR (
3085|             | L"Reboot needed\n" ); break;
3086|         case PSM_ERROR_NOT_INSTALLED: WRITE_ERROR (
3087|             | L"PSM Not installed\n" ); break;
3088|         case STATUS_SUCCESS:          break;
3089|         default:                      WRITE_ERROR (
3090|             | L"Error %08x getting version\n", status ); break;
3091|     }
3092|
3093|     if(status==STATUS_SUCCESS) {
3094|         ULONG VersionHi=(version.Version & 0xff00) >>
3095|             | 8;
3096|         ULONG VersionLo= version.Version & 0x00ff;
3097|         ULONG BuildNum=(version.Version >> 16) &
3098|             | 0x0fff;
3099|
3100|         WRITE ( L"LoVersion = 0x%08lx\n",
3101|             | version.LoVersion );
3102|         WRITE ( L"Version  = %d.%02x build %d\n",
3103|             | VersionHi, VersionLo, BuildNum);
3104|         WRITE ( L"Eval    = %s\n", version.Eval ?
3105|             | L"yes":L"no");
3106|
3107|         if(version.Eval) {
3108|             SYSTEMTIME SysTime={0};
3109|             FILETIME NewFileTime;
3110|
3111|             | FileTimeToLocalFileTime((PFILETIME)&version.ExpireDate,&
3112|             | NewFileTime);
3113|
3114|             | FileTimeToSystemTime(&NewFileTime,&SysTime);
3115|
3116|             if(version.Expired) {
3117|                 WRITE (L"Expired on %d/%02d/%4d

```

```

    | %d:%02d\n",
3108|         SysTime.wMonth,
3109|         SysTime.wDay,
3110|         SysTime.wYear,
3111|         SysTime.wHour,
3112|         SysTime.wMinute);
3113|     } else {
3114|         WRITE(L"Expires on %d/%02d/%4d
    | %d:%02d\n",
3115|         SysTime.wMonth,
3116|         SysTime.wDay,
3117|         SysTime.wYear,
3118|         SysTime.wHour,
3119|         SysTime.wMinute);
3120|     }
3121| }
3122| WRITE(L"\n");
3123| }
3124|
3125| if(status) {
3126|     PrintWin32Error(status);
3127| }
3128|
3129| return 0;
3130| }
3131|
3132| //-----
    | -----
3133|
3134| ULONG RevertSnapShotToPristineState ( ULONG SnapShot )
3135| {
3136|     ULONG Err=0;
3137|     Err = Psm_RevertSnapShotToPristineState(
        | (PVOID)SnapShot );
3138|     if(!Err) {
3139|         WRITE(L"Snapshot %08x is restored to it's
        | pristine state.\n",SnapShot);
3140|     } else {
3141|         WRITE_ERROR(L"Error %08x restoring snapshot
        | %08x to it's pristine state.\n",Err,SnapShot);
3142|     }
3143|     if(Err) {
3144|         PrintWin32Error(Err);
3145|     }
3146|     return Err;
3147| }
3148|
3149| //-----
    | -----
3150|

```

```

3151| ULONG RevertToSnapShot (
3152|     ULONG   SnapShot,
3153|     ULONG   RevertFlags )
3154| {
3155|     WCHAR ch=0;
3156|     ULONG Err=0;
3157|
3158|     if(!AlwaysDelete) {
3159|         WRITE(L"This is a destructive operation.
3160|         | Continue (Y/N)? ");
3161|         do {
3162|             ch = (WCHAR) toupper(getch());
3163|             } while((ch!='Y') && (ch!='N'));
3164|             WRITE(L"%c\n",ch);
3165|         } else {
3166|             ch='Y';
3167|         }
3168|
3169|         if(ch=='Y') {
3170|             Err = Psm_RevertToSnapShot( (PVOID)SnapShot,
3171|             | RevertFlags );
3172|
3173|             if ( Err == PSM_ERROR_REBOOT_NEEDED ) {
3174|                 WRITE_ERROR(L"Revert operation will be
3175|                 | performed at next boot.\n");
3176|             } else if ( Err ==
3177|             | PSM_MULTI_VOLUME_REVERT_DISABLED ) {
3178|                 WRITE_ERROR(L"Multi-volume revert is
3179|                 | disabled - no revert performed.\n");
3180|             } else if(Err) {
3181|                 WRITE_ERROR(L"Error %08x reverting snapshot
3182|                 | %08x\n",Err,SnapShot);
3183|                 PrintWin32Error(Err);
3184|             }
3185|         }
3186|         return Err;
3187|     }
3188|
3189|     //-----
3190|     | -----
3191|
3192|     ULONG ConvertTypeToFlags (
3193|         ULONG   SnapShotType,
3194|         BYTE    *SnapShotFlags )
3195|     {
3196|         ULONG Err = 0;
3197|
3198|         if ( SnapShotFlags ) {
3199|             switch ( SnapShotType ) {

```

```

3194|         case PSM_SS_FLAG_HEADER_READWRITE:
3195|             *SnapShotFlags =
3196|             | PSM_SS_FLAG_P_READWRITE;
3197|             break;
3198|         case PSM_SS_FLAG_HEADER_READONLY:
3199|             *SnapShotFlags =
3200|             | PSM_SS_FLAG_P_READONLY;
3201|             break;
3202|         case
3203|             | PSM_SS_FLAG_HEADER_READWRITE_PERSISTENT:
3204|             *SnapShotFlags =
3205|             | PSM_SS_FLAG_P_READWRITE_PVW;
3206|             break;
3207|         case PSM_SS_FLAG_HEADER_TEMP_READONLY:
3208|             *SnapShotFlags =
3209|             | PSM_SS_FLAG_T_READONLY;
3210|             break;
3211|         case PSM_SS_FLAG_HEADER_TEMP_READWRITE:
3212|             *SnapShotFlags =
3213|             | PSM_SS_FLAG_T_READWRITE;
3214|             break;
3215|         default:
3216|             *SnapShotFlags = 0;
3217|             Err = ERROR_INVALID_PARAMETER;
3218|             break;
3219|     }
3220| } else {
3221|     Err = ERROR_INVALID_PARAMETER;
3222| }
3223| return Err;
3224| }
3225|
3226| //-----
3227| | -----
3228| ULONG ModifySnapShotFlags (
3229|     ULONG SnapShot,
3230|     CHAR Flags )
3231| {
3232|     ULONG Err=0;
3233|     tPSM_GetKernelSnapShotInfoW Get = {0};
3234|     tPSM_SetKernelSnapShotInfo Put = {0};
3235|
3236|     if(SnapShot) {

```

```

3237|     Err =
    | Psm_GetKernelSnapShotInfoW((PVOID)SnapShot,&Get);
3238|     if(!Err) {
3239|         Put.CallerPrivateUse =
    | Get.CallerPrivateUse;
3240|         Put.NumToKeep      = Get.NumToKeep;
3241|         Put.Priority       = Get.Priority;
3242|         Put.Reserved1      = Get.Reserved1;
3243|         Put.Reserved2      = Get.Reserved2;
3244|
3245|         // We want to change what type of snapshot
    | (temp/persist, readonly/readwrite),
3246|         // but leave any other bits (e.g. save on
    | exit) alone.
3247|         Put.SnapShotFlags = Get.SnapShotFlags;
3248|         Put.SnapShotFlags &=
    | ~PSM_SS_FLAG_TYPE_MASK;
3249|         Put.SnapShotFlags |= (Flags &
    | PSM_SS_FLAG_TYPE_MASK);
3250|
3251|         Err = Psm_SetKernelSnapShotInfoW (
    | (PVOID)SnapShot, &Put );
3252|         if ( Err ) {
3253|             WRITE_ERROR(L"Error %08x setting flags
    | for snapshot %08x\n",Err,SnapShot);
3254|         }
3255|     } else {
3256|         WRITE_ERROR(L"Error %08x getting
    | information for snapshot %08x\n",Err,SnapShot);
3257|     }
3258| }
3259|
3260| return Err;
3261| }
3262|
3263| //-----
    | -----
3264|
3265| /*****
    | *****/
3266| *
3267| * FilerGetVersion()
3268| *
3269| * Given an item from a File ListBox, GetVersion
    | retrieves the version
3270| * information from the specified file.
3271| *
3272| * input:  lpszFileName  -  the name of the file.
3273| *         dwBuffSize    -  > 0 size of buffer to
    | hold version info

```

```

3274| *      szBuff      -  buffer to hold version
    | info
3275| *
3276| *  returns: TRUE if successful, FALSE otherwise.
3277| *
3278| *
3279| \*****
    | *****/
3280| #define NUM_VERSION_INFO_KEYS 13
3281| #define VERSION_INFO_KEY_ROOT  L"\\StringFileInfo\\"
3282|
3283| // .EXE version information key structure: for
    | FilerGetVersion() in DRVPROC.C
3284| typedef struct _VersionKeyInfo {
3285|     WCHAR const *szKey;
3286|     WCHAR      *szValue;
3287| } VKINFO, *LPVKINFO;
3288|
3289| BOOL FilerGetVersion(LPWSTR lpszFileName, DWORD
    | dwBuffSize, LPWSTR szBuff, VKINFO *Info)
3290| {
3291|     //
3292|     // NUM_VERSION_INFO_KEYS should be set to the
    | number of entries in
3293|     // VersionKeys[].
3294|     //
3295|     CONST static WCHAR  *VersionKeys[] = {
3296|         L"ProductName",
3297|         L"ProductVersion",
3298|         L"OriginalFilename",
3299|         L"FileDescription",
3300|         L"FileVersion",
3301|         L"CompanyName",
3302|         L"LegalCopyright",
3303|         L"LegalTrademarks",
3304|         L"InternalName",
3305|         L"PrivateBuild",
3306|         L"SpecialBuild",
3307|         L"Build",
3308|         L"Comments"
3309|     };
3310|
3311|     static WCHAR szNull[1] = L"";
3312|     LPVOID lpInfo;
3313|     DWORD  cch;
3314|     UINT   i;
3315|     WCHAR  key[80];
3316|     WCHAR  lpBuffer[10];
3317|
3318|     GetFileVersionInfoW(lpszFileName, 0, dwBuffSize,

```

```

    | (LPVOID)szBuff );
3319|     swprintf(lpBuffer, L"%04X",
    | GetUserDefaultLangID());
3320|     wcscat(lpBuffer,L"04B0");
3321|     for (i = 0; i < NUM_VERSION_INFO_KEYS; i++) {
3322|         wcscpy(key, VERSION_INFO_KEY_ROOT);
3323|         wcscat(key, lpBuffer);
3324|         wcscat(key, L"\\");
3325|         wcscat(key, VersionKeys[i]);
3326|         Info[i].szKey = VersionKeys[i];
3327|
3328|         //
3329|         // If version info exists, and the key query is
    | successful, add
3330|         // the value. Otherwise, the value for the
    | key is NULL.
3331|         //
3332|         if( dwBuffSize && VerQueryValueW(szBuff, key,
    | &lpInfo, &cch) )
3333|             Info[i].szValue = lpInfo;
3334|         else
3335|             Info[i].szValue = szNull;
3336|     }
3337|
3338|     return TRUE;
3339| }
3340|
3341| ULONG GetVersionOfFile( WCHAR *FileName, VKINFO **Info
    | )
3342| {
3343|     ULONG dwHandle;
3344|     ULONG dwLength = GetFileVersionInfoSizeW( FileName,
    | &dwHandle);
3345|
3346|     if(dwLength) {
3347|         *Info=
    | malloc((dwLength+sizeof(VKINFO)*NUM_VERSION_INFO_KEYS));
3348|         if(*Info) {
3349|
    | FilerGetVersion(FileName,dwLength,(WCHAR*)((char*)*Info
    | )+sizeof(VKINFO)*NUM_VERSION_INFO_KEYS),*Info);
3350|             return 0;
3351|         } else {
3352|             return GetLastError();
3353|         }
3354|     } else {
3355|         *Info = NULL;
3356|         return GetLastError();
3357|     }
3358| }

```

```

3359|
3360| ULONG FreeVersionInfo( VKINFO *Info )
3361| {
3362|     if ( Info ) {
3363|         free(Info);
3364|     }
3365|     return 0;
3366| }
3367|
3368| void DumpInfoAboutFile( WCHAR *FileName )
3369| {
3370|     VKINFO *Info = NULL;
3371|     HANDLE Handle = INVALID_HANDLE_VALUE;
3372|     WIN32_FIND_DATAW Dir = {0};
3373|     WCHAR Buffer[1024];
3374|     WCHAR *p = NULL;
3375|
3376|     if(SearchPathW(NULL,FileName,NULL,1024,Buffer,&p))
3377|     | {
3378|         Handle = FindFirstFileW(Buffer,&Dir);
3379|         if(Handle!=INVALID_HANDLE_VALUE) {
3380|             FILETIME NewFileTime={0};
3381|             SYSTEMTIME SystemTime={0};
3382|
3383|             GetVersionOfFile(FileName,&Info);
3384|
3385|             | FileTimeToLocalFileTime((FILETIME*)&Dir.ftLastWriteTime,
3386|             | &NewFileTime);
3387|
3388|             | FileTimeToSystemTime(&NewFileTime,&SystemTime);
3389|
3390|             WRITE(L"%2d/%02d/%04d %2d:%02d %-15.15s
3391|             | '%s'\n",
3392|             |
3393|             | SystemTime.wMonth,
3394|             | SystemTime.wDay,
3395|             | SystemTime.wYear,
3396|             | SystemTime.wHour,
3397|             | SystemTime.wMinute,
3398|             | Info ? Info[4].szValue : L"",
3399|             | Buffer
3400|             | );
3401|
3402|             FreeVersionInfo(Info);
3403|             FindClose(Handle);
3404|         } else {
3405|             DEBUG_WRITE(L"Unable to get directory
3406|             | information for file '%s'\n",FileName);
3407|         }
3408|     } else {
3409|         DEBUG_WRITE(L"Unable to find file

```



```

    | '%s\n",FileName);
3403|    }
3404| }
3405|
3406| //-----
    | -----
3407|
3408| void PrintHeader( const char *exeName )
3409| {
3410|     WRITE(L"%S - Snapshot Command line management
    | utility\n",exeName);
3411|     WRITE(L"Copyright (c) 2000-2001 Columbia Data
    | Products, Inc. All Rights Reserved.\n\n");
3412| }
3413|
3414| void PrintVersion( const char *exeName )
3415| {
3416|     WCHAR Dir[1024];
3417|
3418|     PrintHeader(exeName);
3419|     WRITE(L"%s version\n",_PsmVendorNameWStr_);
3420|     IsInstalled();
3421|
3422|     WRITE(L"Date/Time      Version      File\n");
3423|
3424|     DumpInfoAboutFile(L"psmlapi.dll");
3425|     DumpInfoAboutFile(L"ss.exe");
3426|
3427|     GetSystemDirectoryW(Dir,1024);
3428|     wcscat(Dir,L"\\psmready.exe");
3429|
3430|     DumpInfoAboutFile(Dir);
3431|
3432|     GetSystemDirectoryW(Dir,1024);
3433|     wcscat(Dir,L"\\drivers\\psman5.sys");
3434|     DumpInfoAboutFile(Dir);
3435|
3436|     GetSystemDirectoryW(Dir,1024);
3437|
    | wcscat(Dir,L"\\serverappliance\\mui\\0409\\snapshot.dll"
    | );
3438|     DumpInfoAboutFile(Dir);
3439|
3440|     GetSystemDirectoryW(Dir,1024);
3441|     wcscat(Dir,L"\\serverappliance\\PSMCom.dll");
3442|     DumpInfoAboutFile(Dir);
3443|
3444|     GetSystemDirectoryW(Dir,1024);
3445|     wcscat(Dir,L"\\serverappliance\\drbackup.dll");
3446|     DumpInfoAboutFile(Dir);

```

```

3447| }
3448|
3449|
3450| void PrintHelp ( const char *exeName )
3451| {
3452|     PrintHeader(exeName);
3453|
3454|     WRITE(
3455|         L" -n          = Create new snapshot (must be
            | followed by one or more -l:x).\n"
3456|         L" -l:x       = Volume to take a snapshot
            | of.\n"
3457|         L" -d:x       = Delete the snapshot specified
            | by x.\n"
3458|         L" -dg:x      = Delete the snapshots in group
            | x.\n"
3459|         L" -deleteall = Delete all snapshots.\n"
3460|         L" -v[s|l]    = List all active snapshots in
            | normal, short or long view.\n"
3461|         L" -stats     = Display cache utilization for
            | all volumes.\n"
3462|         L" -i         = Install PSM.\n"
3463|         L" -u         = Uninstall PSM.\n"
3464|         L" -g:x       = Group number of this
            | snapshot.\n"
3465|         L" -p:x       = Priority, 0=none, 1=lowest,
            | 254=highest, 255=keep always.\n"
3466|         L" -k:x       = Number of snapshots to keep in
            | this group.\n"
3467|         L" -f:x       = Snapshot flags, do '-? flags'
            | for more information.\n"
3468|         L" -name:x    = Name of snapshot, do '-? name'
            | for more information.\n"
3469|         L" -quiet[:full] = Don't display console
            | messages. \n"
3470|         L" -y         = Always assume yes on
            | questions.\n"
3471|         L" -reset:x   = Restore a Read/Write snapshot
            | to its pristine state.\n"
3472|         L" -revert:x  = Restore snapshot x images to
            | original volumes.\n"
3473|         L" -version   = List versions of support
            | modules\n"
3474|         L" -chron     = Sort snapshot list in forward
            | chronological order\n"
3475|         L" -readonly:x = Make snapshot x be read-only
            | (causes reset to pristine state)\n"
3476|         L" -readwrite:x = Make snapshot x be
            | writeable\n"
3477| #if DR_BACKUP_SUPPORTED

```

```

3478|     L" -image:x      = Perform image backup of
    | volume x\n"
3479|     L" -imagetest:x = Fully test correctness of
    | volume image backup in path x (slow)\n"
3480|     L" -imagecheck:x = Quickly test that all files
    | in backup path x exist\n"
3481|     L" -backuptest  = Verify registry settings for
    | disaster recovery backup\n"
3482|     L" -diskette    = Create disaster recovery
    | diskette\n"
3483| #endif /*DR_BACKUP_SUPPORTED*/
3484|     L" -createcache = Create cache files on
    | volume(s) specified by '-l:x ...'\n"
3485| );
3486| }
3487|
3488| void PrintHelpName ( const char *exeName )
3489| {
3490|     PrintHeader(exeName);
3491|
3492|     WRITE(
3493|         L" The following can be used as substitution
    | strings for snapshot names:\n\n"
3494|         L" %%i = Instance number (Unique per
    | snapshot).\n"
3495|         L" %%M = Month (1-12).\n"
3496|         L" %%D = Day (1-31).\n"
3497|         L" %%Y = Year (2000-20xx).\n"
3498|         L" %%m = Minute (0-59).\n"
3499|         L" %%h = Hour in 12 hour format (1-12).\n"
3500|         L" %%H = Hour in 24 hour format (0-23).\n"
3501|         L" %%s = Second (0-59).\n"
3502|         L" %%w = Three letter weekday abbreviation
    | (Sun, Mon, etc..).\n"
3503|         L" %%W = Full weekday name (Sunday, Monday,
    | etc..).\n"
3504|         L" %%a = AM/PM marker.\n"
3505|         L" %%p = Group number.\n"
3506|         L" %%P = Priority (0-255).\n"
3507|         L" %%k = Number of snapshots to keep for this
    | group.\n"
3508|         L" %%%% = Percent sign.\n"
3509|         L"\nExamples:\n"
3510|         L" '-name:Hourly' or "
3511|         L" '-name:Snapshot at %%H:%%m\n"
3512|         L"\nThe above characters are case-sensitive,
    | and any duplicate names are postfixed\n"
3513|         L"with an unique number.\n"
3514|     );
3515| }

```

```

3516|
3517| void PrintHelpFlags ( const char *exeName )
3518| {
3519|     PrintHeader(exeName);
3520|
3521|     WRITE(
3522|         L" Any one of the following snapshot flags can
        | be set:\n\n"
3523|         L" 0 = Virtual volumes are read-write, with
        | changes discarded after reboot.\n"
3524|         L" 1 = Virtual volumes are read-only.\n"
3525|         L" 2 = Virtual volumes are read-write, with
        | persistent writes.\n"
3526|         L" 3 = Virtual volumes are temporary and
        | read-only.\n"
3527|         L" 4 = Virtual volumes are temporary and
        | read-write.\n"
3528|         L"\nExamples:\n"
3529|         L" '-f:1' or '-f:2'\n"
3530|     );
3531| }
3532|
3533| PSMSTATUS PSMAPI Psm_TestFunction( ULONG Param1, ULONG
        | Param2);
3534|
3535| void DoJobTest( ULONG Param1, ULONG Param2 )
3536| {
3537|     #ifdef _DEBUG
3538|         Psm_TestFunction(Param1,Param2);
3539|     #endif
3540| }
3541|
3542|
3543| //-----
        | -----
3544|
3545| #ifdef _DEBUG
3546| void FixNtVolumeName ( WCHAR *FileName )
3547| {
3548|     //
3549|     // FixNtVolumeName() turns '\\?\Volume{...}' into
        | '\??\Volume{...}'.
3550|     // This is special voodoo that makes things work
        | real good.
3551|     //
3552|     const WCHAR *GuidPattern = L"\\?\Volume{";
3553|     const ULONG GuidPatternChars = wcslen(GuidPattern);
3554|     if (
        | wcsncmp(FileName,GuidPattern,GuidPatternChars)==0 ) {
3555|         FileName[1] = L'?';

```

```

3556|  }
3557| }
3558| #endif /*_DEBUG*/
3559|
3560| //-----
| -----
3561|
3562| #ifdef _DEBUG
3563| BOOLEAN DeleteCacheFilesDuringCleanup = FALSE;
3564| BOOLEAN ShouldDeletePsmFile ( const WCHAR *FileName )
3565| {
3566|     //
3567|     //  ShouldDeletePsmFile() usually returns TRUE,
| meaning that the file given
3568|     //  should be deleted.  It is called only with PSM
| filenames.
3569|     //  If the user did not specify '-cleanup:cache',
| however, it
3570|     //  returns FALSE when it sees a difference file.
| Keeping the cache files
3571|     //  by default makes things go much quicker when
| we create the first snapshot later.
3572|     //
3573|     BOOLEAN Nukelt = TRUE;
3574|
3575|     if ( !DeleteCacheFilesDuringCleanup ) {
3576|         const WCHAR *TailPattern = L".diff.psm";
3577|         ULONG TailPatternChars = wcslen(TailPattern);
3578|         ULONG FileNameChars = wcslen(FileName);
3579|         if ( FileNameChars >= TailPatternChars ) {
3580|             if (
| wcsicmp(&FileName[FileNameChars-TailPatternChars],TailPa
| ttern) == 0 ) {
3581|                 Nukelt = FALSE;
3582|             }
3583|         }
3584|     }
3585|
3586|     return Nukelt;
3587| }
3588| #endif /*_DEBUG*/
3589|
3590| //-----
| -----
3591|
3592| #ifdef _DEBUG
3593| int DeleteAllPsmFiles()
3594| {
3595|     //
3596|     //  DeleteAllPsmFiles() looks in the

```

```

    | FilesNotToBackup key in the registry
3597|    // for entries put there by PSM. This is a
    | convenient way to get a list
3598|    // of all the PSM file names on the system.
3599|    //
3600|    int Err = 0;
3601|    int i = 0;
3602|    const ULONG ValueDataChars = 16384; // in case we
    | put multiple filenames in MULTI_SZ
3603|    const ULONG ValueDataBytes = sizeof(WCHAR) *
    | ValueDataChars;
3604|    DWORD ValueDataBytesReturned = 0;
3605|    WCHAR *RegPath =
    | L"SYSTEM\\CurrentControlSet\\Control\\BackupRestore\\Fil
    | esNotToBackup";
3606|    HKEY KeyHandle = INVALID_HANDLE_VALUE;
3607|    WCHAR ValueName [256] = {0};
3608|    DWORD ValueNameCharCount = 0;
3609|    DWORD ValueType = 0;
3610|    WCHAR *ValueData = (WCHAR *)
    | malloc(ValueDataBytes);
3611|    const WCHAR *FrontPattern = L"Persistent Storage
    | Manager (";
3612|    const WCHAR *MiddlePatterns[] = { L"Cache:",
    | L"Header:", L"Index:", 0 };
3613|    const ULONG FrontPatternChars =
    | wcslen(FrontPattern);
3614|
3615|    if ( ValueData ) {
3616|        DWORD KeyErr = RegOpenKeyExW (
3617|            HKEY_LOCAL_MACHINE,
3618|            RegPath,
3619|            0,
3620|            KEY_READ,
3621|            &KeyHandle );
3622|
3623|        // Look in FilesNotToBackup list for values
    | that begin with
3624|        // "PersistentStorageManager (Cache:", "...
    | (Header:", or "... (Index:".
3625|        // The value of each such key will be a PSM
    | filename.
3626|
3627|        if ( KeyErr == ERROR_SUCCESS ) {
3628|            // Enumerate all values within the key we
    | just opened...
3629|
3630|            for ( i=0; KeyErr == ERROR_SUCCESS; ++i ) {
3631|                ValueNameCharCount = sizeof(ValueName)
    | / sizeof(ValueName[0]);

```

```

3632|         ValueDataBytesReturned =
| ValueDataBytes;
3633|         KeyErr = RegEnumValueW(
3634|             KeyHandle,          //
| handle to key to query
3635|             i,                  //
| index of value to query
3636|             ValueName,          //
| value buffer
3637|             &ValueNameCharCount, //
| size of value buffer (IN/OUT)
3638|             NULL,              //
| reserved
3639|             &ValueType,        //
| type buffer
3640|             (unsigned char *)ValueData, //
| data buffer
3641|             &ValueDataBytesReturned ); //
| size of data buffer (IN/OUT)
3642|
3643|         if ( KeyErr == ERROR_SUCCESS ) {
3644|             if ( ValueType == REG_MULTI_SZ ) {
3645|                 if (
| wcsncmp(ValueName,FrontPattern,FrontPatternChars) == 0
| ) {
3646|                     // Okay, this value was
| added by PSM, but is it really a PSM filename list?
3647|                     WCHAR *Middle = ValueName +
| FrontPatternChars;
3648|                     BOOLEAN FoundPattern =
| FALSE;
3649|                     int pi;
3650|                     for ( pi=0;
| MiddlePatterns[pi]; ++pi ) {
3651|                         ULONG
| MiddlePatternChars = wcslen(MiddlePatterns[pi]);
3652|                         if (
| wcsncmp(Middle,MiddlePatterns[pi],MiddlePatternChars)==0
| ) {
3653|                             FoundPattern =
| TRUE; // Yes, it is a list of PSM filenames.
3654|                             break;
3655|                         }
3656|                     }
3657|
3658|                     if ( FoundPattern ) {
3659|                         ULONG DelStatus = 0;
3660|                         WCHAR *FileName =
| ValueData;
3661|                         while ( *FileName ) {

```

```

3662|
3663|     | FixNtVolumeName(FileName);
3664|     | ShouldDeletePsmFile(FileName) ) {
3665|         DelStatus =
3666|         | Psm_DeletePsmFileW (FileName);
3667|         if ( DelStatus
3668|         | == 0 ) {
3669|             | WRITE(L"Deleted file '%s'\n", FileName);
3670|         } else {
3671|             | //WRITE(L">>> Psm_DeletePsmFile('%s') returned
3672|             | %08x\n",FileName,DelStatus);
3673|         }
3674|     } else {
3675|         | WRITE(L"Retained file '%s'\n",FileName);
3676|     }
3677|     FileName += 1 +
3678|     | wcslen(FileName); // skip to next string in MULTI_SZ
3679| }
3680| }
3681| }
3682| }
3683| }
3684| }
3685| }
3686| }
3687| }
3688| }
3689| }
3690| }
3691| }
3692| #endif /*_DEBUG*/
3693|
3694| //-----
3695| | -----
3696| typedef struct sKeyNameListNode {
3697|     struct sKeyNameListNode *Next;
3698|     WCHAR *KeyName;
3699| } tKeyNameListNode, *pKeyNameListNode;
3700|
3701| // FIXFIXFIX: This function should have an option to

```



```

    | determine whether
3702| // child keys should be deleted, not just the values in
    | those keys.
3703| // Will need to do something similar to the logic for
    | enumerating and
3704| // putting names in linked list, then going back and
    | deleting everything
3705| // in the linked list. This is because enumerator
    | functions have undefined
3706| // behavior if the registry tree is being modified at
    | the same time.
3707|
3708| void RecursiveKeyDelete (
3709|     HKEY          parent,          //
    | handle of key to search for values to delete
3710|     const WCHAR    *parentName,     //
    | keeps track of where we are for display
3711|     const WCHAR    *listOfValuesToDelete[] ) //
    | array of names for the registry values to delete
3712| {
3713|     //
3714|     // RecursiveKeyDelete() looks through the given
    | registry key 'parent'
3715|     // for any of the values whose names are specified
    | in NULL-terminated
3716|     // array of wide strings 'listOfValuesToDelete'.
    | After that, it recursively
3717|     // opens all child keys of parent, one at a time,
    | and calls recursively
3718|     // to delete any of the targeted values
    | underneath.
3719|     //
3720|     WCHAR keyName [256];
3721|     const DWORD keyNameChars = sizeof(keyName) /
    | sizeof(WCHAR);
3722|     WCHAR *childName = malloc(sizeof(WCHAR) *
    | (keyNameChars + wcslen(parentName) + 1));
3723|     int i = 0;
3724|     FILETIME lastWriteTime = {0};
3725|     LONG KeyStatus = 0;
3726|     if ( childName ) {
3727|         // First try to delete each of the keys we are
    | told to kill,
3728|         // whether or not they are really there.
3729|
3730|         if ( listOfValuesToDelete ) {
3731|             for ( i=0; listOfValuesToDelete[i]; ++i ) {
3732|                 KeyStatus = RegDeleteValueW ( parent,
    | listOfValuesToDelete[i] );
3733|                 swprintf ( childName, L"%s\\%s",

```

```

    | parentName, listOfValuesToDelete[i] );
3734|         if ( KeyStatus == ERROR_SUCCESS ) {
3735|             WRITE(L"Deleted registry value
    | '%s'\n", childName);
3736|         } else {
3737|             DEBUG_WRITE(L">>>
    | RegDeleteKeyW('%s') returned %08x\n", childName,
    | KeyStatus);
3738|         }
3739|     }
3740| } else {
3741|     // listOfValuesToDelete==NULL. This means
    | delete all values in the key.
3742|     pKeyNameListNode valueNameList = NULL;
3743|     WCHAR valueNameBuffer [256];
3744|     KeyStatus = 0;
3745|     for ( i=0; ; ++i )
3746|     {
3747|         DWORD valueNameChars =
    | sizeof(valueNameBuffer) / sizeof(valueNameBuffer[0]);
3748|         KeyStatus = RegEnumValueW (
3749|             parent,          // handle to
    | key to query
3750|             i,                // index of
    | value to query
3751|             valueNameBuffer,  //
    | value buffer
3752|             &valueNameChars, // size of
    | value buffer
3753|             NULL,             // reserved
3754|             NULL,             // type buffer
3755|             NULL,             // data buffer
3756|             NULL );           // size of data
    | buffer
3757|
3758|         if ( KeyStatus == 0 ) {
3759|             pKeyNameListNode newNode =
    | (pKeyNameListNode) malloc (sizeof(tKeyNameListNode));
3760|             if ( newNode ) {
3761|                 newNode->KeyName =
    | wcsdup(valueNameBuffer);
3762|                 if ( newNode->KeyName ) {
3763|                     newNode->Next =
    | valueNameList;
3764|                     valueNameList = newNode;
3765|                 } else {
3766|                     free (newNode);
3767|                     WRITE_ERROR(L"Out of memory
    | in RecursiveKeyDelete\n");
3768|                     break;

```

```

3769|         }
3770|     } else {
3771|         WRITE_ERROR(L"Out of memory in
    | RecursiveKeyDelete\n");
3772|         break;
3773|     }
3774| } else {
3775|     break; // no more values in this
    | key
3776| }
3777| }
3778|
3779| while ( valueNameList ) {
3780|     pKeyNameListNode next =
    | valueNameList->Next;
3781|     KeyStatus = RegDeleteValueW ( parent,
    | valueNameList->KeyName );
3782|     swprintf ( childName, L"%s\\%s",
    | parentName, valueNameList->KeyName );
3783|     if ( KeyStatus == 0 ) {
3784|         DEBUG_WRITE(L">>> Deleted registry
    | value '%s'\n", childName);
3785|     } else {
3786|         WRITE_ERROR(L"Error %08x trying to
    | delete registry value '%s'\n",KeyStatus,childName);
3787|     }
3788|     free (valueNameList->KeyName);
3789|     free (valueNameList);
3790|     valueNameList = next;
3791| }
3792| }
3793|
3794| // Now recursively traverse subkeys...
3795|
3796| KeyStatus = ERROR_SUCCESS;
3797| for ( i=0; KeyStatus == ERROR_SUCCESS; ++i ) {
3798|     DWORD keyNameCharCount = keyNameChars;
3799|     KeyStatus = RegEnumKeyExW (
3800|         parent,
3801|         i,
3802|         keyName,
3803|         &keyNameCharCount,
3804|         NULL,
3805|         NULL,
3806|         NULL,
3807|         &lastWriteTime );
3808|
3809|     if ( KeyStatus == ERROR_SUCCESS ) {
3810|         HKEY childKey = INVALID_HANDLE_VALUE;
3811|         DWORD OpenStatus = 0;

```

```

3812|         OpenStatus = RegOpenKeyExW ( parent,
    | keyName, 0, KEY_ALL_ACCESS, &childKey );
3813|         if ( OpenStatus == ERROR_SUCCESS ) {
3814|             swprintf ( childName, L"%s\\%s",
    | parentName, keyName );
3815|             RecursiveKeyDelete ( childKey,
    | childName, listOfValuesToDelete );
3816|             RegCloseKey (childKey);
3817|         }
3818|     }
3819| }
3820|
3821|     free (childName);
3822| }
3823| }
3824|
3825| //-----
    | -----
3826|
3827| int CleanSakFromRegistry()
3828| {
3829|     const WCHAR * const ParentNameList[] = {
3830|         // EventFilter
3831|         | L"software\\Microsoft\\ServerAppliance\\EventFilter\\Eve
    | nts\\EventPSMWarnings",
3832|         | L"software\\Microsoft\\ServerAppliance\\EventFilter\\Eve
    | nts\\EventPSMErrors",
3833|
3834|         // ElementManager
3835|         | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\TabsDisksPSM",
3836|         | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\TabsDisksPSMSetup",
3837|         | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\TabsDisksPSMSetupList",
3838|         | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\TabsDisksPSMSched",
3839|         | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\TabsDisksPSMPI",
3840|         | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\TabsDisksPSMSysBackup",
3841|

```

| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\TabsDisksPSMDisaster",  
3842|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\TabsDisksPSMVolRestore",  
3843|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\AlertDefinitionsPSMEvent1",  
3844|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMSchedList",  
3845|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMSetupList",  
3846|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMSchedNew",  
3847|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMSchedDelete",  
3848|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMSchedProp",  
3849|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMSetup",  
3850|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMSetupVols",  
3851|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIDelete",  
3852|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIList",  
3853|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPINew",  
3854|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIProp",  
3855|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIUndo",  
3856|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMVRList",  
3857|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMVRRestore",

3858|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIContext",  
3859|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIBUStart",  
3860|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIBUStop",  
3861|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIBUMkDisk",  
3862|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIBUStatus",  
3863|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\ContextHelpPSMPIBUProp",  
3864|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCDiskPSM",  
3865|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMSettingUp",  
3866|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMManagingImages",  
3867|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMManagingSchedules",  
3868|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMUsingVolRestore",  
3869|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMUsingDR",  
3870|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMVolumeStatus",  
3871|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMVolumeConfig",  
3872|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMNewImage",  
3873|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\  
| WebElementDefinitions\\HelpTOCPSMDeleteImage",  
3874|  
| L"software\\Microsoft\\ServerAppliance\\ElementManager\\

```

    | WebElementDefinitions\\HelpTOCPSMUndoWrites",
3875|
    | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\HelpTOCPSMImageProperties",
3876|
    | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\HelpTOCPSMImageContext",
3877|
    | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\HelpTOCPSMNewSchedule",
3878|
    | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\HelpTOCPSMDeleteSchedule",
3879|
    | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\HelpTOCPSMScheduleProperties",
3880|
    | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\HelpTOCPSMRestoringImageSet",
3881|
    | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\HelpTOCPSMChangeRecoverySettings"
    | ,
3882|
    | L"software\\Microsoft\\ServerAppliance\\ElementManager\\
    | WebElementDefinitions\\HelpTOCPSMMakeDiskette",
3883|
3884|    // The End!
3885|    NULL
3886| };
3887|
3888| HKEY Key = INVALID_HANDLE_VALUE;
3889| int i = 0;
3890|
3891| for ( i=0; ParentNameList[i] != NULL; ++i ) {
3892|     ULONG Err = RegOpenKeyExW ( HKEY_LOCAL_MACHINE,
    | ParentNameList[i], 0, KEY_ALL_ACCESS, &Key );
3893|     if ( Err == 0 ) {
3894|         RecursiveKeyDelete ( Key,
    | ParentNameList[i], NULL );
3895|         RegCloseKey(Key);
3896|         Key = INVALID_HANDLE_VALUE;
3897|         Err = RegDeleteKeyW ( HKEY_LOCAL_MACHINE,
    | ParentNameList[i] );
3898|         if ( Err == 0 ) {
3899|             DEBUG_WRITE(L">>> Deleted key '%s\\n",
    | ParentNameList[i] );
3900|         } else {
3901|             WRITE_ERROR(L"Error %08x deleting key
    | '%s\\n", Err, ParentNameList[i]);

```

```

3902|     }
3903|   } else {
3904|     if ( Err != 2 ) {
3905|       DEBUG_WRITE(L">>> Error %08x opening
| registry key '%s'\n",Err,ParentNameList[i]);
3906|     }
3907|   }
3908| }
3909|
3910| return 0;
3911| }
3912|
3913| //-----
| -----
3914|
3915| #ifdef _DEBUG
3916| int DeleteFileMapRegistryKeys()
3917| {
3918|   //
3919|   // DeleteFileMapRegistryKeys() looks through a
| list of registry keys
3920|   // for places where we store file maps, and
| deletes them.
3921|   // File maps tell the PSM driver which clusters on
| disk are used by
3922|   // various PSM files so that it can do direct I/O
| to them before the
3923|   // filesystem is mounted. We need to erase these
| whenever the PSM files
3924|   // themselves are erased, so that PSM doesn't try
| to write directly to disk
3925|   // in places it shouldn't.
3926|   //
3927|   int Err = 0;
3928|   int i = 0;
3929|
3930|   const WCHAR *listOfValuesToDelete[] = {
| L"HeaderMap", L"CacheMap", L"IndexMap", 0 };
3931|   const WCHAR *ParentNameList[] = {
3932|     L"SYSTEM\\CurrentControlSet\\Services\\PSMan5",
3933|     L"Cluster\\PersistentStorageManager",
3934|     0
3935|   };
3936|
3937|   for ( i=0; ParentNameList[i]; ++i ) {
3938|     HKEY ParentKey = INVALID_HANDLE_VALUE;
3939|     DWORD KeyErr = RegOpenKeyExW (
3940|       HKEY_LOCAL_MACHINE,
3941|       ParentNameList[i],
3942|       0,

```



```

3943|         KEY_ALL_ACCESS,
3944|         &ParentKey );
3945|
3946|         if ( KeyErr == ERROR_SUCCESS ) {
3947|             RecursiveKeyDelete ( ParentKey,
3948|             | ParentNameList[i], listOfValuesToDelete );
3949|             RegCloseKey (ParentKey);
3950|         }
3951|     }
3952|     return Err;
3953| }
3954| #endif /*_DEBUG*/
3955|
3956| //-----
3957| | -----
3958| #ifdef _DEBUG
3959| int ResetPsmState()
3960| {
3961|     //
3962|     // ResetPsmState() called by '-cleanup' option.
3963|     // If '-cleanup:cache' was used, the global
3964|     | variable DeleteCacheFilesDuringCleanup
3965|     // will already have been set to TRUE (or FALSE
3966|     | if just '-cache').
3967|     // By default, we don't delete cache files so it
3968|     | doesn't take forever to
3969|     // create a snapshot afterward.
3970|     //
3971|     int Err = 0;
3972|     WCHAR ch = 'Y';
3973|
3974|     if ( !AlwaysDelete ) {
3975|         WRITE(L"All snapshots and PSM files will be
3976|         | deleted.\n");
3977|         if ( !DeleteCacheFilesDuringCleanup ) {
3978|             WRITE(L"(Difference files will be retained
3979|             | for quick snapshot creation.)\n");
3980|         }
3981|         WRITE(L"Continue (Y/N)? ");
3982|         do {
3983|             ch = (WCHAR)toupper(getch());
3984|         } while((ch!='Y') && (ch!='N'));
3985|         WRITE(L"%c\n",ch);
3986|     }
3987|
3988|     if ( ch == 'Y' ) {
3989|         Err = DeleteAllExistingSnapShots();
3990|         if ( Err == 0 ) {

```

```

3986|         Err = DeleteAllPsmFiles();
3987|         if ( Err == 0 ) {
3988|             Err = DeleteFileMapRegistryKeys();
3989|         }
3990|     }
3991| }
3992|
3993|     return Err;
3994| }
3995| #endif /*_DEBUG*/
3996|
3997| //-----
| -----
3998|
3999| #define JOB_HELP                0
4000| #define JOB_NEW                  1
4001| #define JOB_DELETE              2
4002| #define JOB_DELETEALL           3
4003| #define JOB_LIST                4
4004| #define JOB_INSTALL             5
4005| #define JOB_UNINSTALL           6
4006| #define JOB_LISTALL             7
4007| #define JOB_STATISTICS          8
4008| #define JOB_ISINSTALLED         9
4009| #define JOB_HELP_NAME           10
4010| #define JOB_HELP_FLAGS          11
4011| #define JOB_DELETE_GROUP        12
4012| #define JOB_REVERT_TO_A_SNAPSHOT 13
4013| #define JOB_REVERT_SNAPSHOT_TO_PRISTINE_STATE 14
4014| #define JOB_VERSION             15
4015| #define JOB_MODIFY_SNAPSHOT_FLAGS 16
4016| #define JOB_CREATE_VOLUME_BACKUP 17
4017| #define JOB_SET_CLUSTER_REGISTRY 18
4018| #define JOB_TURBO_SCRUB         19
4019| #define JOB_GET_GLOBAL_STATUS    20
4020| #define JOB_TEST_BACKUP_IMAGE_FULL 21
4021| #define JOB_TEST_BACKUP_IMAGE_QUICK 22
4022| #define JOB_CREATE_RECOVERY_DISKETTE 23
4023| #define JOB_TEST_BACKUP_PATHS    24
4024| #define JOB_ABORT_BACKUP         25
4025| #define JOB_CREATE_CACHE         26
4026| #define JOB_CLEAN_SAK           27
4027| #define JOB_TEST_BOOTINI         28
4028|
4029| #ifdef _DEBUG
4030| #define JOB_TEST 999
4031| #endif
4032|
4033| //-----
| -----

```

```

4034|
4035| void SetJobToDo (
4036|     ULONG *JobToDo,
4037|     ULONG WhatToDo,
4038|     const char *JobCommandOption,
4039|     const char *ExeName )
4040| {
4041|     static const char *PrevJobOption = NULL;
4042|
4043|     if ( PrevJobOption ) {
4044|         fprintf ( stderr,
4045|             "Error: Cannot specify both '%s' and '%s'
4046|             | in the same run of %s\n\n",
4047|             PrevJobOption,
4048|             JobCommandOption,
4049|             ExeName );
4050|         PrintHelp(ExeName);
4051|
4052|         exit(1);
4053|     } else {
4054|         PrevJobOption = JobCommandOption;
4055|         *JobToDo = WhatToDo;
4056|     }
4057| }
4058|
4059|
4060| #define SET_JOB(Job)
4061|     | SetJobToDo(&JobToDo,(Job),ThisOption,ansi_argv[0])
4062| //undocumented psm calls
4063| PSMSTATUS PSMAPI Psm_SetClusterRegistry( PVOID SnapShot
4064|     | );
4065| int _cdecl main ( int ansi_argc, char *ansi_argv[] )
4066| {
4067|     int i = 0;
4068|     int errorlevel = 0;
4069|     BOOLEAN GotList=FALSE;
4070|     ULONG JobToDo = JOB_HELP;
4071|     ULONG Err = 0;
4072|     int argc = 0;
4073|     LPWSTR *argv = 0;
4074|     ULONG Snapshot = 0;
4075|     int SnapshotRepeat = 1; // undocumented! (for
4076|         | tracking down a weird bug)
4077|     ULONG View=VIEW_NORMAL;
4078|     const char *ThisOption = NULL;
4079|     #ifdef _DEBUG
4080|     ULONG Param1 = 0;

```

```

4080|    ULONG Param2 = 0;
4081| #endif
4082|    ULONG RevertFlags =
4083|        PSM_REVERT_FLAG_CREATE_UNDO_SNAPSHOT |
4084|        PSM_REVERT_FLAG_DISMOUNT_AFFECTED_VOLUMES;
4085|
4086|    //----- begin stuff for "-image"
    | -----
4087|    const WCHAR *VolName = NULL;
4088|    //----- end stuff for "-image"
    | -----
4089|
4090|    SetErrorMode ( SEM_FAILCRITICALERRORS |
    | SEM_NOOPENFILEERRORBOX );
4091|
4092|    argv = CommandLineToArgvW(GetCommandLineW(),
    | &(argc) );
4093|
4094|    for(i=1;i<argc && errorlevel==0;i++) {
4095|        char *end = 0;
4096|        ThisOption = ansi_argv[i];
4097|        if(stricmp(ThisOption,"-n")==0) {
4098|            SET_JOB (JOB_NEW);
4099|        } else if(stricmp(ThisOption,"-version")==0) {
4100|            SET_JOB (JOB_VERSION);
4101| #ifdef _DEBUG
4102|        } else if(stricmp(ThisOption,"-test")==0) {
4103|            Param1 = strtoul(ansi_argv[++i],&end,16);
4104|            Param2 = strtoul(ansi_argv[++i],&end,16);
4105|            SET_JOB (JOB_TEST);
4106| #endif
4107|        } else if(stricmp(ThisOption,"-quiet")==0) {
4108|            QuietMode = TRUE;
4109|        } else if(stricmp(ThisOption,"-debug")==0) {
4110|            DebugEnabled = TRUE;
4111|        } else if(stricmp(ThisOption,"-y")==0) {
4112|            AlwaysDelete = TRUE;
4113|        } else if(stricmp(ThisOption,"-quiet:full")==0)
        | {
4114|            QuietMode = TRUE;
4115|            QuietModeError = TRUE;
4116|        } else if(stricmp(ThisOption,"-d")==0) {
4117|            // this option is for backwards
            | compatabilty, use -d: instead
4118|            Snapshot = strtoul(ansi_argv[++i],&end,16);
4119|            SET_JOB (JOB_DELETE);
4120|        } else if(strnicmp(ThisOption,"-d:",3)==0) {
4121|            Snapshot = strtoul(ansi_argv[i]+3,&end,16);
4122|            SET_JOB (JOB_DELETE);
4123|        } else if(strnicmp(ThisOption,"-scr:",5)==0) {

```

```

4124|     Snapshot = strtoul(ansi_argv[i]+5,&end,16);
4125|     SET_JOB (JOB_SET_CLUSTER_REGISTRY);
4126| } else if(strncmp(ThisOption,"-dg:",4)==0) {
4127|     GroupNumber =
| strtoul(ansi_argv[i]+4,&end,16);
4128|     SET_JOB (JOB_DELETE_GROUP);
4129| } else if(strncmp(ThisOption,"-reset:",7)==0)
| {
4130|     Snapshot = strtoul(ansi_argv[i]+7,&end,16);
4131|     SET_JOB
| (JOB_REVERT_SNAPSHOT_TO_PRISTINE_STATE);
4132| } else if(strncmp(ThisOption,"-revert:",8)==0)
| {
4133|     Snapshot = strtoul(ansi_argv[i]+8,&end,16);
4134|     SET_JOB (JOB_REVERT_TO_A_SNAPSHOT);
4135| } else if(stricmp(ThisOption,"-noundo")==0) {
4136|     RevertFlags &=
| ~PSM_REVERT_FLAG_CREATE_UNDO_SNAPSHOT;
4137| } else if(stricmp(ThisOption,"-atboot")==0) {
4138|     RevertFlags |= PSM_REVERT_FLAG_ATBOOT;
4139| } else
| if(stricmp(ThisOption,"-abortbackup")==0) {
4140|     SET_JOB(JOB_ABORT_BACKUP);
4141| } else if(strncmp(ThisOption,"-name:",6)==0) {
4142|     wcsncpy(PatternName,argv[i]+6);
4143| } else if(strncmp(ThisOption,"-repeat",7)==0)
| {
4144|     int numScanned =
| sscanf(ThisOption,"-repeat:%i",&SnapshotRepeat);
4145|     if ( numScanned != 1 ) {
4146|         SnapshotRepeat = 3;
4147|     }
4148|     WRITE ( L"Just set snapshot repeat counter
| to %d\n", SnapshotRepeat);
4149| } else if(stricmp(ThisOption,"-listall")==0 ) {
4150|     // This undocumented job type added to test
| new PSMLAPI functions.
4151|     SET_JOB (JOB_LISTALL); // list both
| persistent and temporary snapshots
4152| } else if(stricmp(ThisOption,"-deleteall")==0)
| {
4153|     SET_JOB (JOB_DELETEALL);
4154| } else if(strncmp(ThisOption,"-v",2)==0) {
4155|     SET_JOB (JOB_LIST);
4156|     if(toupper(ThisOption[2]) == 'S') {
4157|         View = VIEW_SHORT;
4158|     } else if(toupper(ThisOption[2]) == 'L') {
4159|         View = VIEW_LONG;
4160|     }
4161| } else if(stricmp(ThisOption,"-chron")==0) {

```

```

4162|         ViewSortOrder = FORWARD_CHRON;
4163|     } else if(stricmp(ThisOption,"-i")==0) {
4164|         SET_JOB (JOB_INSTALL);
4165|     } else if(stricmp(ThisOption,"-status")==0) {
4166|         SET_JOB (JOB_GET_GLOBAL_STATUS);
4167|     } else if(stricmp(ThisOption,"-u")==0) {
4168|         SET_JOB (JOB_UNINSTALL);
4169|     } else
4170|         | if(stricmp(ThisOption,"-isinstalled")==0) {
4171|         SET_JOB (JOB_ISINSTALLED);
4172|     } else
4173|         | if(stricmp(ThisOption,"-createcache")==0) {
4174|         SET_JOB (JOB_CREATE_CACHE);
4175|     } else if(strnicmp(ThisOption,"-g:",3)==0) {
4176|         GroupNumber = strtoul(ThisOption+3,&end,0);
4177|     } else if(strnicmp(ThisOption,"-f:",3)==0) {
4178|         ULONG SnapShotType =
4179|         | strtoul(ThisOption+3,&end,0);
4180|         if ( ConvertTypeToFlags(SnapShotType,
4181|         | &SnapShotFlags) != 0 ) {
4182|             WRITE_ERROR(L"Error: Invalid snapshot
4183|             | type %d after '-f:\n",SnapShotType);
4184|             errorlevel = 1;
4185|         }
4186|     } else
4187|         | if(stricmp(ThisOption,"-discardtemp")==0) {
4188|         SaveTempOnExit = FALSE;
4189|         DEBUG_WRITE(L">>> main: Turning off
4190|         | SaveTempOnExit\n");
4191|     } else if(strnicmp(ThisOption,"-p:",3)==0) {
4192|         Priority =
4193|         | (BYTE)(strtoul(ThisOption+3,&end,0) & 0xff);
4194|     } else if(strnicmp(ThisOption,"-k:",3)==0) {
4195|         NumToKeep = strtoul(ThisOption+3,&end,0);
4196|     } else if(strnicmp(ThisOption,"-l:",3)==0) {
4197|         ULONG Continue=FALSE;
4198|
4199|         if(strlen(ThisOption)==4) {
4200|             CHAR Ch=0;
4201|             ULONG c =
4202|             | sscanf(ThisOption+3,"%c",&Ch);
4203|             if (c == 1) {
4204|                 Ch = (char) toupper(Ch);
4205|
4206|                 //WRITE(L"Adding Volume %c\n",Ch);
4207|                 InVolumeMapData[NumVolumes][0] =
4208|                 | Ch;
4209|                 InVolumeMapData[NumVolumes][1] =
4210|                 | ':';
4211|                 InVolumeMapData[NumVolumes][2] =

```

```

    | "\\\';
4201|         Continue = TRUE;
4202|     } else {
4203|         goto OPTIONL_ERROR;
4204|     }
4205| } else
4206|     if(strlen(ThisOption)>4) {
4207|         WRITE(L"Adding Volume
    | '%s\n",argv[i]+3);
4208|
    | wcsncpy(InVolumeMapData[NumVolumes],argv[i]+3);
4209|         Continue = TRUE;
4210|     } else {
4211| OPTIONL_ERROR:
4212|         WRITE_ERROR ( L"Invalid syntax with
    | parameter '%s\n", ThisOption );
4213|         errorlevel = 1;
4214|     }
4215|
4216|     if(Continue) {
4217|
    | if(Psm_CanBePSMedW(InVolumeMapData[NumVolumes])) {
4218|
    | InVolumeMap[NumVolumes]=InVolumeMapData[NumVolumes];
4219|         VolumeMapFlags[NumVolumes] =
    | PSM_VOLUME_MAP;
4220|         NumVolumes++;
4221|         GotList=TRUE;
4222|     } else {
4223|
    | if(Psm_IsAnPSMVolumeW(InVolumeMapData[NumVolumes])) {
4224|         WRITE_ERROR(L"Skipping '%s' as
    | it is an psm volume\n",InVolumeMapData[NumVolumes]);
4225|     } else {
4226|         WRITE_ERROR(L"Skipping '%s' as
    | it is not a local hard
    | drive\n",InVolumeMapData[NumVolumes]);
4227|     }
4228|     }
4229| } else {
4230|     }
4231| } else if(strnicmp(ThisOption,"-stats:",7)==0)
    | {
4232|     WRITE_ERROR(L"This command is obsolete,
    | please use '-stats -l:x' instead\n\n");
4233| } else if(strnicmp(ThisOption,"-stats",6)==0) {
4234|     SET_JOB(JOB_STATISTICS);
4235| } else
    | if(strnicmp(ThisOption,"-readwrite:",11)==0) {
4236|     Snapshot = strtoul(ThisOption+11,&end,16);

```

```

4237|         SnapShotFlags =
| PSM_SS_FLAG_P_READWRITE_PVW;
4238|         SET_JOB (JOB_MODIFY_SNAPSHOT_FLAGS);
4239|     } else
| if(strnicmp(ThisOption,"-readonly:",10)==0) {
4240|         Snapshot = strtoul(ThisOption+10,&end,16);
4241|         SnapShotFlags = PSM_SS_FLAG_P_READONLY;
4242|         SET_JOB (JOB_MODIFY_SNAPSHOT_FLAGS);
4243|     } else if(stricmp(ThisOption,"-?")==0) {
4244|         if(i+1<argc) {
4245|             if(wcsicmp(argv[i+1],L"name")==0) {
4246|                 SET_JOB (JOB_HELP_NAME);
4247|             } else
4248|             if(wcsicmp(argv[i+1],L"flags")==0) {
4249|                 SET_JOB (JOB_HELP_FLAGS);
4250|             } else {
4251|                 SET_JOB (JOB_HELP);
4252|             }
4253|             i++;
4254|         } else {
4255|             SET_JOB (JOB_HELP);
4256|         }
4257|     } else
| if(strnicmp(ThisOption,"-imagecheck:",12)==0 ) {
4258|         SET_JOB (JOB_TEST_BACKUP_IMAGE_QUICK);
4259|         VolName = &argv[i][12];
4260|         if ( !*VolName ) {
4261|             WRITE_ERROR(L"Error: expected backup
| path after '-imagecheck:\n");
4262|             errorlevel = 1;
4263|         }
4264|     } else
| if(strnicmp(ThisOption,"-imagetest:",11)==0 ) {
4265|         SET_JOB (JOB_TEST_BACKUP_IMAGE_FULL);
4266|         VolName = &argv[i][11];
4267|         if ( !*VolName ) {
4268|             WRITE_ERROR(L"Error: expected backup
| path after '-imagetest:\n");
4269|             errorlevel = 1;
4270|         }
4271|     } else if(strnicmp(ThisOption,"-image:",7)==0)
| {
4272|         SET_JOB (JOB_CREATE_VOLUME_BACKUP);
4273|         VolName = &argv[i][7];
4274|         if ( !*VolName ) {
4275|             WRITE_ERROR ( L"Error: expected volume
| name after '-image:\n" );
4276|             errorlevel = 1;
4277|         }
4278|     } else if(stricmp(ThisOption,"-diskette")==0) {

```



```

4279|         SET_JOB (JOB_CREATE_RECOVERY_DISKETTE);
4280|     } else if(stricmp(ThisOption,"-backuptest")==0)
4281|     | {
4281|         SET_JOB (JOB_TEST_BACKUP_PATHS);
4282|     } else if(stricmp(ThisOption,"-cleansak")==0) {
4283|         SET_JOB (JOB_CLEAN_SAK);
4284| #ifdef _DEBUG
4285|     } else if(strnicmp(ThisOption,"-cleanup",8)==0
4286|     | ) {
4286|         SET_JOB (JOB_TURBO_SCRUB);
4287|         if ( ThisOption[8]=='.' ) {
4288|             if ( stricmp(ThisOption+9,"cache")==0 )
4289|             | {
4289|                 DeleteCacheFilesDuringCleanup =
4290|                 | TRUE;
4290|             } else {
4291|                 WRITE_ERROR ( L"Error: unknown
4292|                 | option '%S' after '-cleanup:\n", ThisOption+9 );
4292|                 errorlevel = 1;
4293|             }
4294|         }
4295| #endif /* _DEBUG */
4296|     } else
4297|     | if(stricmp(ThisOption,"-testbootini")==0) {
4297|         SET_JOB (JOB_TEST_BOOTINI);
4298|     } else {
4299|         WRITE_ERROR ( L"Error: Unknown command
4300|         | line option '%S'\n", ThisOption );
4300|         errorlevel = 1;
4301|     }
4302| }
4303|
4304| // Psm_Register needs to be called before calling
4305| | any
4305| // Psm_Enable* functions.
4306|
4307| if ( errorlevel == 0 ) {
4308|     if ( JobToDo == JOB_HELP ) {
4309|         PrintHelp (ansi_argv[0]);
4310|         errorlevel = 1;
4311|     } else if ( JobToDo == JOB_HELP_NAME ) {
4312|         PrintHelpName (ansi_argv[0]);
4313|         errorlevel = 1;
4314|     } else if ( JobToDo == JOB_HELP_FLAGS ) {
4315|         PrintHelpFlags (ansi_argv[0]);
4316|         errorlevel = 1;
4317|     } else {
4318|         Err = Psm_RegisterW(L"Columbia Data
4319|         | Products, Inc",    // Company name
4319|         L"ss.exe - Persistent

```

```

    | snapshot command line", // Product name
4320|             NULL,
    | // Version number
4321|             PSM_CODE_CDP_TPSM,
    | // Enabling code
4322|
    | PSM_CODE_CDP_TPSM_LICENSE // Key code
4323|     );
4324|
4325|     if(!Err) {
4326|         switch(JobToDo) {
4327|             case JOB_NEW:
4328|                 if(GotList) {
4329|                     int i;
4330|                     for ( i=0; i<SnapshotRepeat
    | && errorlevel==0; ++i ) {
4331|                         errorlevel =
    | MakeNewSnapShot();
4332|                     }
4333|                 } else {
4334|                     WRITE_ERROR ( L"No volumes
    | were specified... use -l:x for each volume x to be
    | PSM'ed\n" );
4335|                     errorlevel = 1;
4336|                 }
4337|                 break;
4338|             case JOB_REVERT_TO_A_SNAPSHOT:
4339|                 errorlevel =
    | RevertToSnapShot(Snapshot, RevertFlags);
4340|                 break;
4341|             case
    | JOB_REVERT_SNAPSHOT_TO_PRISTINE_STATE:
4342|                 errorlevel =
    | RevertSnapShotToPristineState(Snapshot);
4343|                 break;
4344|             case JOB_CREATE_CACHE:
4345|                 errorlevel = CreateCache();
4346|                 break;
4347|             case JOB_VERSION:
4348|                 PrintVersion(ansi_argv[0]);
4349|                 break;
4350| #ifdef _DEBUG
4351|             case JOB_TEST:
4352|                 DoJobTest(Param1,Param2);
4353|                 break;
4354| #endif
4355|             case JOB_DELETE:
4356|                 errorlevel =
    | DeleteExistingSnapShot(Snapshot);
4357|                 break;

```

```

4358|         case JOB_DELETE_GROUP:
4359|             errorlevel =
4360|             | DeleteAllSnapShotsInGroup(GroupNumber);
4361|             break;
4362|         case JOB_DELETEALL:
4363|             errorlevel =
4364|             | DeleteAllExistingSnapShots();
4365|             break;
4366|         case JOB_LIST:
4367|             errorlevel =
4368|             | ListExistingSnapShots(View);
4369|             break;
4370|         case JOB_LISTALL:
4371|             errorlevel = TestList();
4372|             break;
4373|         case JOB_INSTALL:
4374|             errorlevel = InstallPsm();
4375|             break;
4376|         case JOB_UNINSTALL:
4377|             errorlevel = UninstallPsm();
4378|             break;
4379|         case JOB_ISINSTALLED:
4380|             errorlevel = IsInstalled();
4381|             break;
4382|         case JOB_SET_CLUSTER_REGISTRY:
4383|             errorlevel =
4384|             | Psm_SetClusterRegistry((pSnapShot)Snapshot);
4385|             if(errorlevel) {
4386|                 | PrintWin32Error(errorlevel);
4387|             }
4388|             break;
4389|         case JOB_ABORT_BACKUP : {
4390|             HANDLE hEvent;
4391|             PSECURITY_DESCRIPTOR sd={0};
4392|             SECURITY_ATTRIBUTES sa={0};
4393|             sa.nLength =
4394|             | sizeof(SECURITY_ATTRIBUTES);
4395|             sa.bInheritHandle = FALSE;
4396|             sa.lpSecurityDescriptor = sd;
4397|             // Create Event
4398|             sd =
4399|             | GetSecuritySD(FALSE,EVENT_ALL_ACCESS,EVENT_ALL_ACCESS,0)
4400|             | ;
4401|             if(sd) {
4402|                 hEvent = CreateEvent(&sa,
4403|                 | TRUE, FALSE, TEXT("Global\\PSM_Backup_Abort_Event" ));
4404|                 if(hEvent) {
4405|                     BOOLEAN

```

```

    | BackupRunning=GetLastError()==ERROR_ALREADY_EXISTS;
4399|                // Set Event
4400|                SetEvent(hEvent);
4401|                // Close Handle
4402|                CloseHandle(hEvent);
4403|
4404|                if(BackupRunning) {
4405|                    WRITE(L"Backup
    | aborted\n");
4406|                } else {
4407|                    WRITE(L"No backup
    | running\n");
4408|                }
4409|            } else {
4410|                WRITE_ERROR(L"Error
    | %08x opening event\n",GetLastError());
4411|            }
4412|            LocalFree(sd);
4413|        } else {
4414|            Err = GetLastError();
4415|            WRITE_ERROR(L"Error %08x
    | getting security\n",Err);
4416|        }
4417|        break;
4418|    }
4419|    case JOB_GET_GLOBAL_STATUS : {
4420|        tPSM_GetStatusOutW Status;
4421|
    | errorlevel=Psm_GetStatusW(&Status,256,PSM_GLOBAL_STATUS)
    | ;
4422|        if(!errorlevel) {
4423|            WRITE(L"        PSM
    | Status = %s\n",Status.StatusMessage);
4424|        } else {
4425|
    | PrintWin32Error(errorlevel);
4426|        }
4427|
    | errorlevel=Psm_GetStatusW(&Status,256,PSM_REVERT_STATUS)
    | ;
4428|        if(!errorlevel) {
4429|            WRITE(L"        Revert
    | Status = %s\n",Status.StatusMessage);
4430|        } else {
4431|
    | PrintWin32Error(errorlevel);
4432|        }
4433| #if DR_BACKUP_SUPPORTED
4434|
    | errorlevel=Psm_GetStatusW(&Status,256,PSM_ENGINE_STATUS)

```

```

| ;
4435|         if(!errorlevel) {
4436|             WRITE(L"Backup Engine
| Status = %s\n",Status.StatusMessage);
4437|         } else {
4438|
| PrintWin32Error(errorlevel);
4439|         }
4440| #endif /*DR_BACKUP_SUPPORTED*/
4441|         break;
4442|     }
4443|     case JOB_STATISTICS:
4444|         errorlevel =
| DisplayPsmStatistics();
4445|         break;
4446|     case JOB_MODIFY_SNAPSHOT_FLAGS:
4447|         errorlevel =
| ModifySnapShotFlags (Snapshot, SnapShotFlags);
4448|         break;
4449|     case JOB_CREATE_VOLUME_BACKUP:
4450|         errorlevel = DoSnapShotBackup
| (VolName);
4451|         break;
4452|     case JOB_TEST_BACKUP_IMAGE_FULL:
4453|         errorlevel = TestImageBackup
| (VolName, FALSE);
4454|         break;
4455|     case JOB_TEST_BACKUP_IMAGE_QUICK:
4456|         errorlevel = TestImageBackup
| (VolName, TRUE);
4457|         break;
4458|     case JOB_CREATE_RECOVERY_DISKETTE:
4459|         errorlevel =
| CreateRecoveryDiskette();
4460|         break;
4461|     case JOB_TEST_BACKUP_PATHS:
4462|         errorlevel = TestBackupPaths();
4463|         break;
4464|     case JOB_CLEAN_SAK:
4465|         errorlevel =
| CleanSakFromRegistry();
4466|         break;
4467|     case JOB_TEST_BOOTINI:
4468|         UpdateBootIni();
4469|         break;
4470| #ifdef _DEBUG
4471|     case JOB_TURBO_SCRUB:
4472|         errorlevel = ResetPsmState();
4473|         break;
4474| #endif /*_DEBUG*/

```

```

4475|
4476|         default:
4477|             WRITE_ERROR(L"Nothing to do\n");
4478|             errorlevel = 1;
4479|             break;
4480|         }
4481|     } else {
4482|         WRITE_ERROR(L"Error %08x registering
| with psm\n",Err);
4483|         errorlevel = 1;
4484|     }
4485| }
4486| }
4487|
4488| if ( errorlevel>255 || errorlevel<0 ) {
4489|     errorlevel = 255;    // ??? Probably the
| largest value supported by DOS/Windows.
4490| }
4491|
4492| return errorlevel;
4493| }
4494|
4495| /*--- end of file ss.c ---*/
4496|
4497|
4498|
4499| File Listing: File: ss.h
4500|
4501| extern ULONG QuietMode;
4502| extern ULONG QuietModeError;
4503| extern ULONG DebugEnabled;
4504|
4505| #define WRITE if(!QuietMode) wprintf
4506| #define WRITE_ERROR if(!QuietModeError) wprintf
4507| #define DEBUG_WRITE if(DebugEnabled) wprintf
4508|
4509| #ifndef NT_SUCCESS
4510|     #define NT_SUCCESS(status) ((status)==0)
4511| #endif
4512|
4513| typedef unsigned long NTSTATUS;
4514|
4515| #define STATUS_SUCCESS ((NTSTATUS)0)
4516|
4517| #ifndef STATIC
4518|     #ifdef _DEBUG
4519|         #define STATIC
4520|     #else
4521|         #define STATIC static
4522|     #endif

```

```

4523| #endif
4524|
4525| //-----
| -----
| --
4526|
4527| void UpdateBootIni();
4528|
4529| //-----
| -----
| --
4530|
4531| /*--- end of file ss.h ---*/
4532|
4533|
4534|
4535| File Listing: File: vimage.c
4536|
4537| #include "precomp.h"
4538|
4539| #define MAX_LOCATIONS      3
4540| #define MAX_IMAGES_PER_LOCATION 9
4541|
4542| const WCHAR * const BackupRegistryPath =
4543|
4544| | L"SYSTEM\\CurrentControlSet\\Services\\PSMan5\\Backup\\"
4545| | ;
4546|
4547| const WCHAR * const EXCLUSION_FILENAME =
4548| | L"Backup_In_Progress";
4549|
4550| /*-----
| -----
4551|
4552| ***** Glossary of Numbers
4553| | *****
4554|
4555| | There are three different kinds of "Number"
4556| | variables in this code.
4557| | Here they are in descending order of hierarchy:
4558| |
4559| | 1. LocationNumber
4560| | We back up a volume to multiple output streams.
4561| | A LocationNumber tells you which of the streams
4562| | we are talking about.
4563| | LocationNumber is in the range 1 ..
4564| | MAX_LOCATIONS.
4565|
4566| 2. BackupNumber
4567| For each backup location, we do the current

```

```

| backup, but we keep
4562|     a maximum number of previously performed
| backups.
4563|     The current backup is always BackupNumber==1.
| Older backups
4564|     are 2, 3, ... MAX_IMAGES_PER_LOCATION.
4565|
4566| 3. ContinuationNumber
4567|     A given backup is broken up into multiple
| files, based on the
4568|     maximum allowed file size. A
| ContinuationNumber tells you which
4569|     file we are on inside a given backup.
| ContinuationNumbers
4570|     start at 1, and have no definite upper limit.
4571|
4572| Putting this all together:
4573|
4574| If 'd:\vimage.psm' is the first backup location,
| then in the file name:
4575|
4576| d:\vimage.psm\Backup.3\image.007
4577|
4578| The LocationNumber is 1, the BackupNumber is 3, and
| the ContinuationNumber is 7.
4579|
4580| -----
| -----*/
4581|
4582| WCHAR GloballImageName [256] = L"image"; //
| subdirectory prefix after location
4583|
4584| typedef struct sRemoteLogonInfo {
4585|     BOOLEAN    LoggedOn;
4586|     WCHAR      Path [256];           //
| L"d:\backup\"
4587|     WCHAR      UserId [128];
4588|     WCHAR      Password [128];
4589| } tRemoteLogonInfo, *pRemoteLogonInfo;
4590|
4591| typedef struct slmageCreationInfo {
4592|     ULONG      MaxSizeInMegabytes;    //
| If zero, no limit. Otherwise, maximum output file size
| in MB.
4593|     ULONG      NumToKeep;
4594|     ULONG      LocationNumber;        //
| which set of files we are on (1..MAX_LOCATIONS)
4595|     WCHAR      ExclusionFileName [256]; //
| name of file we keep open during backup (delete when
| done)

```



```

4596|  HANDLE          ExclusionFileHandle;
4597|  tRemoteLogonInfo  LogonInfo;
4598| } tImageCreationInfo, *pImageCreationInfo;
4599|
4600| typedef struct sImageOutput {
4601|  HANDLE          Handle;
4602|  ULONG           ByteCounter;
4603|  ULONG           MegabyteCounter;
4604|  ULONG           ImageStatus;
4605|  ULONG           ContinuationNumber;  //
      | which file within a set we are on
4606|  WCHAR           FileName [256];      //
      | current filename (changes on each continuation)
4607|  tImageCreationInfo  OpenInfo;
4608| } tImageOutput, *pImageOutput;
4609|
4610| typedef struct sBackupManager {
4611|  pVolumeImage_Chunk  ChunkBuffer;
4612|  ULONG               ChunkBufferSize;
4613|  ULONG               CumulativeChecksum;
4614|  ULONG               ChunkCounter;
4615|  ULONG               NonFatalError;
4616|  ULONG               NumImageOutputs;
4617|  PVOID               BackupAbortEvent;
4618|  tImageOutput        ImageOutputArray
      | [MAX_LOCATIONS];
4619| } tBackupManager, *pBackupManager;
4620|
4621| typedef struct sLocalLogonInfo {
4622|  WCHAR              UserName[256];
4623|  WCHAR              Password[256];
4624|  ULONG              LogonResult;
4625|  ULONG              ImpersonateResult;
4626|  HANDLE              UserToken;
4627| } tLocalLogonInfo, *pLocalLogonInfo;
4628|
4629| //-----
      | -----
4630|
4631| STATIC tBackupManager  TheBackupManager = {0};
4632|
4633| //-----
      | -----
4634|
4635| typedef ULONG (* BATCH_FILE_WRITER_FUNC) (
4636|  FILE              *BatchFile,
4637|  const WCHAR * const      BatchFilePath,
4638|  const tImageCreationInfo * const  LocationArray,
4639|  ULONG              NumLocations );
4640|

```

```

4641| //-----
    | -----
4642|
4643| ULONG BackupManager_CreateVolumeImage (
4644|     pBackupManager    Manager,
4645|     int                NumOutputFiles,
4646|     plmageCreationInfo OutputFileNames,
4647|     const WCHAR        *VolumeName,
4648|     const WCHAR        *OriginalVolumeName,
4649|     PVOID              AbortEvent );
4650|
4651| ULONG GetTimeStamp ( WCHAR *ts ); // get current time
    | in string of the form 'yyyy,mm,dd,hh,mm,ss'
4652|
4653| ULONG UpdateTimeStamp (
4654|     const WCHAR * const RegPath,
4655|     const WCHAR * const ValueName,
4656|     const WCHAR * const TimeStamp );
4657|
4658| ULONG UpdateEngineStatus ( ULONG Status );
4659| ULONG UpdatePathStatus      ( int LocationNumber,
    | ULONG Status );
4660| ULONG UpdateLastBackupResult ( int LocationNumber,
    | ULONG Status );
4661| ULONG UpdateBackupCount      ( int LocationNumber,
    | ULONG BackupCount );
4662|
4663| ULONG OpenImageOutputFile ( plmageOutput FileInfo );
4664| ULONG CloseImageOutputFile ( plmageOutput FileInfo );
4665|
4666| //-----
    | -----
4667|
4668|
4669| void AppendBackslashIfNeeded ( WCHAR *string )
4670| {
4671|     int len = wcslen(string);
4672|     if ( len>0 && string[len-1]!='\\' ) {
4673|         string[len++] = L'\\';
4674|         string[len] = L'\0';
4675|     }
4676| }
4677|
4678| //-----
    | -----
4679|
4680| typedef struct sShareReference {
4681|     struct sShareReference *Next;
4682|     WCHAR                  *ShareName;
4683|     int                     Count;

```

```

4684| } tShareReference, *pShareReference;
4685|
4686| STATIC pShareReference ShareReferenceList = NULL;
4687|
4688| STATIC pShareReference LocateShareReference ( const
    | WCHAR *ShareName )
4689| {
4690|     pShareReference p;
4691|     for ( p=ShareReferenceList; p; p = p->Next ) {
4692|         if ( _wcsicmp(ShareName,p->ShareName)==0 ) {
4693|             return p;
4694|         }
4695|     }
4696|
4697|     return NULL;
4698| }
4699|
4700| //-----
    | -----
4701|
4702| STATIC ULONG ReferenceLogonShare ( const WCHAR
    | *ShareName )
4703| {
4704|     ULONG Err = 0;
4705|     pShareReference p =
        | LocateShareReference(ShareName);
4706|     if ( p ) {
4707|         ++(p->Count);
4708|     } else {
4709|         p = (pShareReference) malloc
            | (sizeof(tShareReference));
4710|         if ( p ) {
4711|             p->ShareName = (WCHAR *)
                | malloc(sizeof(WCHAR)*(1 + wcslen(ShareName)));
4712|             if ( p->ShareName ) {
4713|                 wcscpy ( p->ShareName, ShareName );
4714|                 p->Next = ShareReferenceList;
4715|                 p->Count = 1;
4716|                 ShareReferenceList = p;
4717|             } else {
4718|                 WRITE_ERROR(L"Out of string memory in
                    | ReferenceLogonShare\n");
4719|             }
4720|         } else {
4721|             WRITE_ERROR(L"Out of node memory in
                    | ReferenceLogonShare\n");
4722|             Err = ERROR_OUTOFMEMORY;
4723|         }
4724|     }
4725|     return Err;

```

```

4726| }
4727|
4728| //-----
    | -----
4729|
4730| STATIC ULONG DereferenceLogonShare ( const WCHAR
    | *ShareName )
4731| {
4732|     ULONG Err = 0;
4733|     pShareReference p =
        | LocateShareReference(ShareName);
4734|     if ( p ) {
4735|         if ( p->Count > 0 ) {
4736|             if ( --(p->Count) == 0 ) {
4737|                 DWORD CancelResult =
                    | WNetCancelConnection2W ( p->ShareName, 0, FALSE );
4738|                 if ( CancelResult == 0 ) {
4739|                     DEBUG_WRITE(L">>> Detached from
                        | share '%s'\n",p->ShareName);
4740|                 } else {
4741|                     WRITE_ERROR(L"Warning: Could not
                        | detach from network share '%s'
                        | (result=%08x)\n",p->ShareName,CancelResult);
4742|                 }
4743|             }
4744|         } else {
4745|             DEBUG_WRITE(L">>> !!!
                | DereferenceLogonShare: Count for '%s' was zero
                | !!!\n",ShareName);
4746|             Err = PSM_ERROR_UNSUCCESSFUL;
4747|         }
4748|     } else {
4749|         Err = PSM_ERROR_NO_SUCH_OBJECT;
4750|         DEBUG_WRITE(L">>> !!! DereferenceLogonShare:
            | Could not find '%s'\n", ShareName);
4751|     }
4752|     return Err;
4753| }
4754|
4755| //-----
    | -----
4756|
4757| STATIC void LogonShareCleanup (void)
4758| {
4759|     pShareReference p = ShareReferenceList;
4760|     while ( p ) {
4761|         pShareReference next = p->Next;
4762|         if ( p->Count > 0 ) {
4763|             DWORD CancelResult = WNetCancelConnection2W
                | ( p->ShareName, 0, FALSE );

```

```

4764|         DEBUG_WRITE(L">>> Canceled connection
| during cleanup (result=%08x) to
| '%s'\n",CancelResult,p->ShareName);
4765|     }
4766|     free (p->ShareName);
4767|     free (p);
4768|     p = next;
4769| }
4770| }
4771|
4772| //-----
| -----
4773|
4774| ULONG GetSettingsForRecoveryDisketteCreation (
| pRemoteLogonInfo LogonInfo )
4775| {
4776|     // This function expects LogonInfo->Path to be a
| valid path.
4777|     // It is where to generate the diskette image.
4778|     // It could be a local path like 'a:\'.
4779|     // Or it could be a remote path like '\\server\share'.
4780|     //
4781|     // The rest of the fields in LogonInfo can be
| garbage on entry.
4782|
4783|     ULONG Err = 0;
4784|     HKEY Key = INVALID_HANDLE_VALUE;
4785|     ULONG DataSize = 0;
4786|     WCHAR RegPath [256] = {0};
4787|
4788|     LogonInfo->LoggedOn = FALSE;
4789|     LogonInfo->Password[0] = 0;
4790|     LogonInfo->UserId[0] = 0;
4791|
4792|     swprintf ( RegPath, L"%s%s", BackupRegistryPath,
| L"Diskette" );
4793|     Err = RegOpenKeyExW (
4794|         HKEY_LOCAL_MACHINE,
4795|         RegPath,
4796|         0,
4797|         KEY_READ,
4798|         &Key );
4799|
4800|     if ( Err == 0 ) {
4801|         DataSize = sizeof(LogonInfo->UserId);
4802|         Err = RegQueryValueExW(
4803|             Key,
4804|             L"LogonUser",
4805|             NULL, // reserved
4806|             NULL, // address of buffer for value type

```

```

4807|         (char*)(LogonInfo->UserId),
4808|         &DataSize );
4809|
4810|     if ( Err == 0 ) {
4811|         DataSize = sizeof(LogonInfo->Password);
4812|         Err = RegQueryValueExW(
4813|             Key,
4814|             L"LogonPassword",
4815|             NULL, // reserved
4816|             NULL, // address of buffer for value
4817|             | type
4818|             (char*)(LogonInfo->Password),
4819|             &DataSize );
4820|         if ( Err == 0 ) {
4821|             // everything worked!
4822|         } else {
4823|             WRITE_ERROR(L"FAIL: Error %08x getting
4824| | value 'LogonPassword' from registry path
4825| | '%s\\n",Err,RegPath);
4826|         }
4827|     } else {
4828|         WRITE_ERROR(L"FAIL: Error %08x getting
4829| | value 'LogonUser' from registry path
4830| | '%s\\n",Err,RegPath);
4831|     }
4832| }
4833|
4834|     RegCloseKey(Key);
4835|     Key = INVALID_HANDLE_VALUE;
4836| } else {
4837|     WRITE_ERROR(L"FAIL: Error %08x opening registry
4838| | key '%s\\n",RegPath);
4839| }
4840|
4841|     DEBUG_WRITE(L">>>
4842| | GetSettingsForRecoveryDisketteCreation returning
4843| | %08x\\n",Err);
4844|     return Err;
4845| }
4846|
4847| //-----
4848| | -----
4849|
4850| ULONG GetSettingsForBackup(
4851|     HANDLE      Parent,
4852|     const WCHAR *Child,
4853|     plmageCreationInfo Info )
4854| {
4855|     ULONG Err = 0;
4856|     HKEY Key = INVALID_HANDLE_VALUE;

```

```

4848|    ULONG    DataSize = 0;
4849|
4850|    wcsncpy(Info->LogonInfo.Path,L "");
4851|    wcsncpy(Info->LogonInfo.UserId,L "");
4852|    wcsncpy(Info->LogonInfo.Password,L "");
4853|    Info->LocationNumber = 0;
4854|    Info->MaxSizeInMegabytes = 1000;
4855|    Info->NumToKeep = 2;
4856|
4857|    Err = RegOpenKeyExW(
4858|        Parent, // handle of open key
4859|        Child, // address of name of subkey to open
4860|        0, // reserved
4861|        KEY_READ, // security access mask
4862|        &Key// address of handle of open key
4863|    );
4864|
4865|    if(Err==0) {
4866|        DataSize = sizeof(Info->LogonInfo.Path);
4867|        Err = RegQueryValueExW(
4868|            Key, // handle of key to query
4869|            L"LocationPath", // address of name of
| value to query
4870|            NULL, // reserved
4871|            NULL, // address of buffer for value type
4872|            (char*)&(Info->LogonInfo.Path), // address
| of data buffer
4873|            &DataSize // address of data buffer size
4874|        );
4875|        if(Err==0) {
4876|            DataSize =
| sizeof(Info->MaxSizeInMegabytes);
4877|            Err = RegQueryValueExW(
4878|                Key, // handle of key to query
4879|                L"LargestFileSizeLocation", // address
| of name of value to query
4880|                NULL, // reserved
4881|                NULL, // address of buffer for value
| type
4882|                (char*)&(Info->MaxSizeInMegabytes), //
| address of data buffer
4883|                &DataSize // address of data buffer
| size
4884|            );
4885|            if(Err==0) {
4886|                DataSize = sizeof(Info->NumToKeep);
4887|                Err = RegQueryValueExW(
4888|                    Key, // handle of key to query
4889|                    L"NumberToKeepLocation", //
| address of name of value to query

```

```

4890|          NULL, // reserved
4891|          NULL, // address of buffer for
| value type
4892|          (char*)&(Info->NumToKeep), //
| address of data buffer
4893|          &DataSize // address of data
| buffer size
4894|      );
4895|      if(Err==0) {
4896|          if ( Info->NumToKeep >
| MAX_IMAGES_PER_LOCATION ) {
4897|              WRITE_ERROR(L"Warning:
| NumberToKeep (%d) is larger than max allowed (%d) -
| decreasing to max
| allowed\n",Info->NumToKeep,MAX_IMAGES_PER_LOCATION);
4898|              Info->NumToKeep =
| MAX_IMAGES_PER_LOCATION;
4899|          } else if ( Info->NumToKeep < 1 ) {
4900|              WRITE_ERROR(L"Warning:
| NumberToKeep (%d) is too small - setting to
| 1.\n",Info->NumToKeep);
4901|              Info->NumToKeep = 1;
4902|          }
4903|
4904|          DataSize =
| sizeof(Info->LogonInfo.UserId);
4905|          Err = RegQueryValueExW(
4906|              Key,
4907|              L"LogonUser",
4908|              NULL,
4909|              NULL,
4910|              (char
| *)&(Info->LogonInfo.UserId),
4911|              &DataSize);
4912|          if ( Err == 0 ) {
4913|              DataSize =
| sizeof(Info->LogonInfo.Password);
4914|              Err = RegQueryValueExW(
4915|                  Key,
4916|                  L"LogonPassword",
4917|                  NULL,
4918|                  NULL,
4919|                  (char*)&(Info->LogonInfo.Password),
4920|                  &DataSize);
4921|              if ( Err == 0 ) {
4922|              } else {
4923|                  WRITE_ERROR(L"Error %08x
| accessing registry for %s LogonPassword\n",Err,Child);
4924|              }

```



```

4925|         } else {
4926|             WRITE_ERROR(L"Error %08x
| accessing registry for %s LogonUser\n",Err,Child);
4927|         }
4928|     } else {
4929|         WRITE_ERROR(L"Error %08x accessing
| registry for %s NumToKeep\n",Err,Child);
4930|     }
4931| } else {
4932|     WRITE_ERROR(L"Error %08x accessing
| registry for %s Largest\n",Err,Child);
4933| }
4934| } else {
4935|     WRITE_ERROR(L"Error %08x accessing registry
| for %s Location\n",Err,Child);
4936| }
4937| // close the registry
4938| RegCloseKey(Key);
4939| } else {
4940|     WRITE_ERROR(L"Error %08x opening registry for
| %s\n",Err,Child);
4941| }
4942| return Err;
4943| }
4944|
4945| //-----
| -----
4946|
4947| BOOLEAN IsAllWhitespace ( const WCHAR *string )
4948| {
4949|     BOOLEAN allWhitespace = TRUE;
4950|     int i;
4951|     for ( i=0; string[i]; ++i ) {
4952|         if ( string[i]!=' ' && string[i]!='\t' &&
| string[i]!='\n' && string[i]!='\r' ) {
4953|             allWhitespace = FALSE;
4954|             break;
4955|         }
4956|     }
4957|
4958|     return allWhitespace;
4959| }
4960|
4961| //-----
| -----
4962|
4963| ULONG ExtractShareNameFromPath (
4964|     const WCHAR    *Path,
4965|     WCHAR          *Share,
4966|     ULONG          ShareSizeInChars )

```

```

4967| {
4968|     ULONG Err = 0;
4969|
4970|     if ( ShareSizeInChars > 0 ) {
4971|         Share[0] = 0;
4972|     }
4973|
4974|     if ( Path[0]=='\\' && Path[1]=='\\' ) {
4975|         // Expect something like
         | L"\\server\share[...]"
4976|         ULONG index = 0; // skip over double
         | backslashes at the front
4977|
4978|         if ( ShareSizeInChars < 2 ) {
4979|             return PSM_ERROR_INSUFFICIENT_BUFFER;
4980|         }
4981|
4982|         Share[index++] = '\\';
4983|         Share[index++] = '\\';
4984|
4985|         // skip to next backslash to delimit server
         | name
4986|         while ( Path[index] && Path[index]!='\\' ) {
4987|             if ( index >= ShareSizeInChars ) {
4988|                 return PSM_ERROR_INSUFFICIENT_BUFFER;
4989|             }
4990|             Share[index] = Path[index];
4991|             ++index;
4992|         }
4993|
4994|         if ( Path[index] == '\\' ) {
4995|             // skip to next backslash or end of string
         | to get share name
4996|             if ( index >= ShareSizeInChars ) {
4997|                 return PSM_ERROR_INSUFFICIENT_BUFFER;
4998|             }
4999|             Share[index] = Path[index];
5000|             ++index;
5001|             while ( Path[index] && Path[index]!='\\' )
         | {
5002|                 if ( index >= ShareSizeInChars ) {
5003|                     return
         | PSM_ERROR_INSUFFICIENT_BUFFER;
5004|                 }
5005|                 Share[index] = Path[index];
5006|                 ++index;
5007|             }
5008|
5009|             if ( index >= ShareSizeInChars ) {
5010|                 return PSM_ERROR_INSUFFICIENT_BUFFER;

```

```

5011|     }
5012|
5013|     Share[index] = 0;
5014|     DEBUG_WRITE(L">>> Extracted '%s' as share
    | name from path '%s\\n", Share, Path);
5015|     } else {
5016|         WRITE_ERROR(L"Error: Share name missing in
    | network path '%s\\n",Path);
5017|         Err = PSM_ERROR_INVALID_PATH;
5018|     }
5019| } else {
5020|     WRITE_ERROR(L"Error: Invalid network path
    | '%s\\n",Path);
5021|     Err = PSM_ERROR_INVALID_PATH;
5022| }
5023|
5024| return Err;
5025| }
5026|
5027| //-----
    | -----
5028|
5029| ULONG ParseServerAndShareFromPath (
5030|     const WCHAR * const    Path,
5031|     WCHAR *          Server,
5032|     WCHAR *          Share,
5033|     WCHAR *          RestOfPath )
5034| {
5035|     ULONG Err = 0;
5036|     int numScanned = 0;
5037|
5038|     numScanned = swscanf (Path,
    | L"\\\\\\%[^\\]\\%[^\\]\\%s",Server,Share,RestOfPath);
5039|     if ( numScanned < 2 ) {
5040|         Err = PSM_ERROR_INVALID_PATH;
5041|     } else {
5042|         if ( numScanned < 3 ) {
5043|             RestOfPath[0] = '\\0';
5044|         }
5045|     }
5046|
5047|     return Err;
5048| }
5049|
5050| //-----
    | -----
5051|
5052| ULONG RemoteLogonCheck ( pRemoteLogonInfo LogonInfo )
5053| {
5054|     ULONG Err = 0;

```

```

5055|  LogonInfo->LoggedOn = FALSE;
5056|  if ( LogonInfo->Path[0]!='\\' &&
    | LogonInfo->Path[1]!='\\' ) {
5057|      // Assume it's a network path.
5058|      // Only log in if they specified a user...
5059|      if ( !IsAllWhitespace(LogonInfo->UserId) ) {
5060|          DWORD LogonResult = 0;
5061|          NETRESOURCEW NetResource = {0};
5062|          WCHAR RemoteName[256] = {0};
5063|
5064|          Err = ExtractShareNameFromPath (
5065|              LogonInfo->Path,
5066|              RemoteName,
5067|
    | sizeof(RemoteName)/sizeof(RemoteName[0]) );
5068|
5069|          if ( Err == 0 ) {
5070|              NetResource.dwType      =
    | RESOURCETYPE_DISK; // we are attaching to a disk
5071|              NetResource.lpLocalName = NULL;
    | // do not redirect any local device name to network
    | resource
5072|              NetResource.lpRemoteName =
    | RemoteName; // "\\server\share"
5073|              NetResource.lpProvider   = NULL;
    | // let Windows search for provider based on remote name
5074|
5075|              Err = WNetAddConnection2W (
5076|                  &NetResource, //
    | connection details
5077|                  LogonInfo->Password, //
    | password
5078|                  LogonInfo->UserId, // user
    | name
5079|                  0 ); //
    | connection options
5080|
5081|          if ( Err == 0 ) {
5082|              DEBUG_WRITE(L">>> Attached to '%s'
    | with user id '%s'\n",RemoteName,LogonInfo->UserId);
5083|              Err = ReferenceLogonShare (
    | RemoteName );
5084|              if ( Err == 0 ) {
5085|                  LogonInfo->LoggedOn = TRUE;
5086|              }
5087|          } else {
5088|              WRITE_ERROR(L"Error %08x attaching
    | to '%s' with user id '%s' (maybe bad
    | password?)\n",Err,RemoteName,LogonInfo->UserId);
5089|          }

```

```

5090|         } else {
5091|             WRITE_ERROR(L"Error %08x extracting
| share name from path '%s'\n",Err,LogonInfo->Path);
5092|         }
5093|     }
5094| } else {
5095|     // Assume it's a local path.
5096| }
5097| return Err;
5098| }
5099|
5100| //-----
| -----
5101|
5102| ULONG RemoteLogoff ( pRemoteLogonInfo LogonInfo )
5103| {
5104|     ULONG Err = 0;
5105|     if ( LogonInfo->LoggedOn ) {
5106|         WCHAR RemoteName [256] = {0};
5107|         Err = ExtractShareNameFromPath (
5108|             LogonInfo->Path,
5109|             RemoteName,
5110|             sizeof(RemoteName) / sizeof(RemoteName[0])
5111|         );
5112|         if ( Err == 0 ) {
5113|             DereferenceLogonShare (RemoteName);
5114|             LogonInfo->LoggedOn = FALSE;
5115|         } else {
5116|             DEBUG_WRITE(L">>> Could not extract share
| from '%s' for canceling connection;
| error=%08x\n",LogonInfo->Path,Err);
5117|         }
5118|     }
5119|     return Err;
5120| }
5121|
5122| //-----
| -----
5123|
5124| ULONG LocalLogonCheck ( pLocalLogonInfo LogonInfo )
5125| {
5126|     // Get userid, password from 'Backup' in
| registry...
5127|     HKEY Key = INVALID_HANDLE_VALUE;
5128|     LogonInfo->ImpersonateResult =
| PSM_ERROR_UNSUCCESSFUL;
5129|     LogonInfo->UserToken = INVALID_HANDLE_VALUE;
5130|     LogonInfo->LogonResult = RegOpenKeyExW(
5131|         HKEY_LOCAL_MACHINE, // handle of open key

```

```

5132|     BackupRegistryPath,    // address of name of
    | subkey to open
5133|     0,                    // reserved
5134|     KEY_READ,              // security access mask
5135|     &Key );                // address of handle of
    | open key
5136|
5137|     if ( LogonInfo->LogonResult == 0 ) {
5138|         ULONG DataSize = sizeof(LogonInfo->UserName);
5139|         LogonInfo->LogonResult = RegQueryValueExW (
5140|             Key,
5141|             L"LoginName",
5142|             NULL, // reserved
5143|             NULL, // address of buffer for value type
5144|             (char*)&(LogonInfo->UserName),
5145|             &DataSize );
5146|
5147|         if ( LogonInfo->LogonResult == 0 ) {
5148|             DataSize = sizeof(LogonInfo->Password);
5149|             LogonInfo->LogonResult = RegQueryValueExW (
5150|                 Key,
5151|                 L"Password",
5152|                 NULL, // reserved
5153|                 NULL, // address of buffer for value
    | type
5154|                 (char*)&(LogonInfo->Password),
5155|                 &DataSize );
5156|
5157|             if ( LogonInfo->LogonResult == 0 ) {
5158|                 if (
    | IsAllWhitespace(LogonInfo->UserName) ) {
5159|                     LogonInfo->LogonResult =
    | PSM_ERROR_INVALID_PARAMETER;
5160|                     DEBUG_WRITE(L">>> LocalLogonCheck:
    | LoginName is all whitespace - skipping local login\n");
5161|                 } else {
5162|                     int DidLogon = 0;
5163|                     DEBUG_WRITE(L">>> LocalLogonCheck:
    | Attempting to locally log on to
    | user='%s\n",LogonInfo->UserName);
5164|                     DidLogon = LogonUserW (
5165|                         LogonInfo->UserName,
5166|                         NULL,
5167|                         LogonInfo->Password,
5168|
    | LOGON32_LOGON_NETWORK_CLEARTEXT,
5169|                         LOGON32_PROVIDER_DEFAULT,
5170|                         &(LogonInfo->UserToken) );
5171|
5172|                     if ( DidLogon ) {

```

```

5173|             int DidImpersonate =
| ImpersonateLoggedOnUser (LogonInfo->UserToken);
5174|             if ( !DidImpersonate ) {
5175|
| LogonInfo->ImpersonateResult = GetLastError();
5176|             DEBUG_WRITE(L">>>
| LogonLocalCheck: ImpersonateLoggedOnUser '%s' failed
| with code
| %08x\n",LogonInfo->UserName,LogonInfo->ImpersonateResult
| );
5177|             } else {
5178|
| LogonInfo->ImpersonateResult = 0;
5179|             DEBUG_WRITE(L">>>
| LogonLocalCheck: Successfully logged in as user
| '%s'\n",LogonInfo->UserName);
5180|             }
5181|             } else {
5182|             LogonInfo->LogonResult =
| GetLastError();
5183|             DEBUG_WRITE(L">>>
| LocalLogonCheck: LogonUser failed with code
| %08x\n",LogonInfo->LogonResult);
5184|             }
5185|             }
5186|             } else {
5187|             DEBUG_WRITE(L">>> LocalLogonCheck:
| Could not get 'Password' registry value\n");
5188|             }
5189|             } else {
5190|             DEBUG_WRITE(L">>> LocalLogonCheck: Could
| not get 'LoginName' registry value\n");
5191|             }
5192|
5193|             RegCloseKey (Key);
5194|             Key = INVALID_HANDLE_VALUE;
5195|             } else {
5196|             DEBUG_WRITE(L">>> LocalLogonCheck: Error %08x
| opening registry key
| '%s'\n",LogonInfo->LogonResult,BackupRegistryPath);
5197|             }
5198|
5199|             return LogonInfo->LogonResult;
5200| }
5201|
5202| //-----
| -----
5203|
5204|
5205| ULONG LocalLogoff ( pLocalLogonInfo LogonInfo )

```

```

5206| {
5207|     ULONG Err = 0;
5208|
5209|     if ( LogonInfo->ImpersonateResult == 0 ) {
5210|         int BackToMyOldSelf = RevertToSelf();
5211|         if ( BackToMyOldSelf ) {
5212|             DEBUG_WRITE(L">>> Successfully reverted to
| self.\n");
5213|         } else {
5214|             Err = GetLastError();
5215|             DEBUG_WRITE(L">>> LocalLogoff: Error %08x
| in RevertToSelf\n",Err);
5216|         }
5217|     }
5218|
5219|     if ( LogonInfo->LogonResult == 0 ) {
5220|         if ( LogonInfo->UserToken !=
| INVALID_HANDLE_VALUE ) {
5221|             CloseHandle (LogonInfo->UserToken);
5222|             LogonInfo->UserToken =
| INVALID_HANDLE_VALUE;
5223|             DEBUG_WRITE(L">> LocalLogoff: Closed user
| token\n");
5224|         }
5225|     }
5226|
5227|     return Err;
5228| }
5229|
5230| //-----
| -----
5231|
5232| ULONG PathExists ( WCHAR *Path )
5233| {
5234|     ULONG Exists = FALSE;
5235|     struct _wfinddatai64_t FindData = {0};
5236|     long hFile = -1;
5237|     int PathLen = wcslen(Path);
5238|     WCHAR *CopyOfPath = (WCHAR *)
| malloc(sizeof(WCHAR)*(PathLen+1));
5239|     if ( CopyOfPath ) {
5240|         wcsncpy ( CopyOfPath, Path );
5241|         if ( PathLen>0 && CopyOfPath[PathLen-1] == '\\'
| ) {
5242|             CopyOfPath[--PathLen] = 0;
5243|         }
5244|         hFile = _wfindfirsti64 ( CopyOfPath, &FindData
| );
5245|         if ( hFile != -1 ) {
5246|             _findclose (hFile);

```



```

5247|         hFile = -1;
5248|         Exists = TRUE;
5249|     }
5250|     free(CopyOfPath);
5251|     CopyOfPath = 0;
5252| } else {
5253|     WRITE_ERROR(L"Error: Out of memory in
    | PathExists\n");
5254| }
5255| DEBUG_WRITE(L">>> PathExists('%s') =
    | %s\n",Path,(Exists?L"TRUE":L"FALSE"));
5256| return Exists;
5257| }
5258|
5259| //-----
    | -----
5260|
5261| ULONG CreateEntirePath ( WCHAR *Path )
5262| {
5263|     ULONG Err = 0;
5264|     ULONG Index = 0;  // index into 'Path' of
    | character after current backslash.
5265|     BOOLEAN Finished = FALSE;
5266|     WCHAR SaveChar = 0;
5267|     int MakeDirResult = 0;
5268|
5269|     DEBUG_WRITE(L">>> CreateEntirePath '%s\n", Path);
5270|
5271|     // The path looks like L"d:\this\is\a\path\" or
    | L"\\server\share\this\is\a\path\"
5272|     // Create "d:\this\", then "d:\this\is\", ...,
    | "d:\this\is\a\path\".
5273|
5274|     if ( Path[0]=='\\' && Path[1]=='\\' ) {
5275|         // Looks like a server path... skip over
    | "\\server\share\"
5276|         DEBUG_WRITE(L">>> CreateEntirePath: looks like
    | server path\n");
5277|         Index = 2;
5278|         while ( Path[Index] && Path[Index]!='\\' ) {
5279|             ++Index;
5280|         }
5281|         if ( Path[Index] == '\\' ) {
5282|             // skipped over "\\server\"; now skip one
    | more backslash.
5283|             while ( Path[Index] && Path[Index]!='\\' )
    | {
5284|                 ++Index;
5285|             }
5286|             if ( !Path[Index] ) {

```

```

5287|         Finished = TRUE;
5288|     }
5289| } else {
5290|     Finished = TRUE;
5291| }
5292| } else {
5293|     // Looks like a local path... just skip over
    | first backslash.
5294|     DEBUG_WRITE(L">>> CreateEntirePath: looks like
    | local path\n");
5295|     Index = 0;
5296|     while ( Path[Index] && Path[Index]!='\\' ) {
5297|         ++Index;
5298|     }
5299|     if ( Path[Index] != '\\' ) {
5300|         Finished = TRUE;
5301|     }
5302| }
5303|
5304| if ( Finished ) {
5305|     Err = PSM_ERROR_INVALID_PATH;
5306|     WRITE_ERROR(L"CreateEntirePath: Path '%s' is
    | too short to be valid!\n",Path);
5307| } else if ( Err == 0 ) {
5308|     ULONG LoopCounter = 0;
5309|     while ( !Finished ) {
5310|         if ( Path[Index] == '\\' ) {
5311|             SaveChar = Path[++Index]; // skip
    | over the backslash.
5312|             Path[Index] = 0;
5313|         } else {
5314|             SaveChar = 0;
5315|         }
5316|
5317|         if ( LoopCounter++ > 0 ) {
5318|             MakeDirResult = _wmkdir(Path);
5319|             DEBUG_WRITE(L">>> _wmkdir('%s')=%d,
    | errno=%d\n", Path, MakeDirResult, errno);
5320|         }
5321|
5322|         if ( SaveChar ) {
5323|             Path[Index] = SaveChar;
5324|             while ( Path[Index] &&
    | Path[Index]!='\\' ) {
5325|                 ++Index;
5326|             }
5327|         } else {
5328|             Finished = TRUE;
5329|         }
5330|     }

```

```

5331|    }
5332|
5333|    DEBUG_WRITE(L">>> CreateEntirePath returning
    | %08x\n",Err);
5334|    return Err;
5335| }
5336|
5337| //-----
    | -----
5338|
5339| ULONG DeleteAllFilesAndDirectory ( const WCHAR *Path )
5340| {
5341|     ULONG Err = 0;
5342|     WCHAR FileSpec [256];
5343|     struct _wfinddatai64_t FindData = {0};
5344|     long hFile = 0;
5345|     int result = 0;
5346|
5347|     DEBUG_WRITE(L">>> DeleteAllFilesAndDirectory:
    | Path='%s\n",Path);
5348|
5349|     wcscpy ( FileSpec, Path );
5350|     AppendBackslashIfNeeded (FileSpec);
5351|     wcscat ( FileSpec, L"*.*" );
5352|     hFile = _wfindfirsti64 ( FileSpec, &FindData );
5353|     if ( hFile != -1 ) {
5354|         do {
5355|             DEBUG_WRITE(L">>> FindFirst/FindNext:
    | '%s\n",FindData.name);
5356|             if ( wcscmp(FindData.name,L".")!=0 &&
    | wcscmp(FindData.name,L"..")!=0 ) {
5357|                 wcscpy ( FileSpec, Path );
5358|                 AppendBackslashIfNeeded (FileSpec);
5359|                 wcscat ( FileSpec, FindData.name );
5360|                 if ( FindData.attrib & _A_SUBDIR ) {
5361|                     DEBUG_WRITE(L">>> Doing recursive
    | call for nested subdirectory...\n");
5362|                     Err = DeleteAllFilesAndDirectory
    | (FileSpec);
5363|                     if ( Err != 0 ) {
5364|                         break;
5365|                     }
5366|                 } else {
5367|                     result = _wremove (FileSpec);
5368|                     if ( result != 0 ) {
5369|                         DEBUG_WRITE(L">>>
    | _wremove('%s')=%d, errno=%d\n",FileSpec,result,errno);
5370|                     }
5371|                 }
5372|             }

```

```

5373|     } while ( _wfindnexti64(hFile,&FindData)==0 &&
    | Err==0 );
5374|     _findclose (hFile);
5375| } else {
5376|     DEBUG_WRITE(L">>> FindFirst failed on
    | '%s\\n",FileSpec);
5377| }
5378|
5379| result = _wrmdir (Path);
5380| if ( result != 0 ) {
5381|     DEBUG_WRITE(L">>> _wrmdir('%s')=%d,
    | errno=%d\\n", Path, result, errno);
5382|     Err = PSM_ERROR_INVALID_PATH;
5383| }
5384|
5385| DEBUG_WRITE(L">>> DeleteAllFilesAndDirectory
    | returning %08x\\n", Err);
5386| return Err;
5387| }
5388|
5389| //-----
    | -----
5390|
5391| ULONG RenameOldBackupPaths (
5392|     const WCHAR    *BackupLocation,
5393|     const WCHAR    *BackupName,
5394|     ULONG          NumToKeep )
5395| {
5396|     // This function takes a path like
    | L"d:\\backup\\path\\" and does the following:
5397|     // (Suppose NumToKeep is 5, and BackupName is
    | L"fred".)
5398|     // Delete the directory (and contained files)
    | L"d:\\backup\\path\\fred.5"
5399|     // Rename L"d:\\backup\\path\\fred.4" to be
    | L"d:\\backup\\path\\fred.5".
5400|     // ...
5401|     // Rename L"d:\\backup\\path\\fred.1" to be
    | L"d:\\backup\\path\\fred.2".
5402|
5403|     ULONG Err = 0;
5404|     WCHAR Path [256];
5405|     WCHAR NewName [256];
5406|     ULONG LastOldBackupNumber = 0;
5407|
5408|     ULONG BackupNumber;    // see "Glossary of
    | Numbers" comments above
5409|     for ( BackupNumber = NumToKeep; Err==0 &&
    | BackupNumber>=1; --BackupNumber ) {
5410|         swprintf ( Path, L"%s%s.%d\\", BackupLocation,

```

```

    | BackupName, BackupNumber );
5411|     if ( BackupNumber == NumToKeep ) {
5412|         DeleteAllFilesAndDirectory (Path);
5413|     } else {
5414|         int RenameResult = 0;
5415|         swprintf ( NewName, L"%s%s.%d\\",
    | BackupLocation, BackupName, 1+BackupNumber );
5416|         RenameResult = _wrename(Path,NewName);
5417|         DEBUG_WRITE(L">>> Rename('%s','%s')=%d,
    | errno=%d\n", Path, NewName, RenameResult, errno);
5418|     }
5419| }
5420|
5421| // If the user decreased the value of NumToKeep, we
    | don't want ancient backups
5422| // sitting around forever. But we don't want to
    | immediately clobber all old
5423| // backups past the NumToKeep if decreasing it was
    | an accident. Therefore,
5424| // we find the last backup in excess of NumToKeep
    | and delete it alone.
5425| // This causes the old backups to disappear
    | gradually, one each time a backup
5426| // happens.
5427|
5428| for ( BackupNumber = 1+NumToKeep; ++BackupNumber
    | ) {
5429|     swprintf ( Path, L"%s%s.%d\\", BackupLocation,
    | BackupName, BackupNumber );
5430|     if ( PathExists(Path) ) {
5431|         LastOldBackupNumber = BackupNumber;
5432|     } else {
5433|         break;
5434|     }
5435| }
5436|
5437| if ( LastOldBackupNumber ) {
5438|     swprintf ( Path, L"%s%s.%d\\", BackupLocation,
    | BackupName, LastOldBackupNumber );
5439|     DEBUG_WRITE(L">>> Killing old backup in path
    | '%s'\n",Path);
5440|     DeleteAllFilesAndDirectory(Path);
5441| }
5442|
5443| return Err;
5444| }
5445|
5446| //-----
    | -----
5447|

```

```

5448| ULONG TestBackupPath ( plmageCreationInfo ImageInfo )
5449| {
5450|     ULONG Err = 0;
5451|     WCHAR TempFileName [256];
5452|     FILE *TempFile = 0;
5453|     int result = 0;
5454|
5455|     // Create a temporary file, then erase it, just to
    | see if we can write
5456|     // to where the next backup will go.
5457|     wcscpy ( TempFileName, ImageInfo->LogonInfo.Path );
5458|     AppendBackslashIfNeeded (TempFileName);
5459|     wscat ( TempFileName, L"vimage.tmp" );
5460|     TempFile = _wfopen(TempFileName,L"wt");
5461|     if ( TempFile ) {
5462|         DEBUG_WRITE(L">>> Successfully opened test file
    | '%s' for write\n", TempFileName);
5463|         fclose(TempFile);
5464|         TempFile = NULL;
5465|         result = _wremove (TempFileName);
5466|         if ( result == 0 ) {
5467|             DEBUG_WRITE(L">>> Successfully deleted test
    | file '%s'\n", TempFileName);
5468|         } else {
5469|             WRITE_ERROR(L"Error: Could not delete test
    | file '%s'\n", TempFileName);
5470|             Err = PSM_ERROR_INVALID_PATH;
5471|         }
5472|     } else {
5473|         WRITE_ERROR(L"Error: Could not write temporary
    | file to path '%s'\n", ImageInfo->LogonInfo.Path);
5474|         Err = PSM_ERROR_INVALID_PATH;
5475|     }
5476|
5477|     return Err;
5478| }
5479|
5480| //-----
    | -----
5481|
5482| ULONG CreateExclusionFile (
5483|     const WCHAR      *BackupPath,
5484|     HANDLE           *ExclusionFileHandle,
5485|     WCHAR            *ExclusionFileName )
5486| {
5487|     ULONG Err = 0;
5488|
5489|     DEBUG_WRITE(L">>> CreateExclusionFile: path='%s',
    | fileptr=%08x\n",BackupPath,*ExclusionFileHandle);
5490|

```

```

5491|  if ( *ExclusionFileHandle != NULL ) {
5492|      Err = PSM_BACKUP_ALREADY_IN_PROGRESS;
5493|      WRITE_ERROR(L"Error: Backup already in
    | progress (in same process)\n");
5494|  } else {
5495|      wcscpy ( ExclusionFileName, BackupPath );
5496|      AppendBackslashIfNeeded ( ExclusionFileName );
5497|      wscat ( ExclusionFileName, EXCLUSION_FILENAME
    | );
5498|      *ExclusionFileHandle = CreateFileW(
5499|          ExclusionFileName,          // file
    | name
5500|          GENERIC_READ | GENERIC_WRITE,  //
    | access mode
5501|          0,                          //
    | share mode
5502|          NULL,                        // SD
5503|          CREATE_ALWAYS,               // how
    | to create
5504|          FILE_ATTRIBUTE_NORMAL,       // file
    | attributes
5505|          NULL );                      //
    | handle to template file
5506|
5507|  if ( *ExclusionFileHandle ==
    | INVALID_HANDLE_VALUE ) {
5508|      // See if the file exists at all...
5509|      ULONG FileError = GetLastError();
5510|      DEBUG_WRITE(L">>> CreateFileW failed:
    | error=%08x\n",FileError);
5511|      if ( PathExists(ExclusionFileName) ) {
5512|          // If the file exists, but we cannot
    | open it, assume a backup is in progress.
5513|          Err = PSM_BACKUP_ALREADY_IN_PROGRESS;
5514|          WRITE_ERROR(L"Error: Backup already in
    | progress (in another process)\n");
5515|      } else {
5516|          // The file does not exist, but we
    | cannot open it; assume invalid path.
5517|          Err = PSM_ERROR_INVALID_PATH;
5518|          WRITE_ERROR(L"Error: Invalid backup
    | path '%s' - cannot create exclusion
    | file\n",BackupPath);
5519|      }
5520|      *ExclusionFileHandle = NULL;
5521|  } else {
5522|      DEBUG_WRITE(L">>> Successfully created
    | exclusion file '%s'\n",ExclusionFileName);
5523|  }
5524|  }

```

```

5525|
5526|     return Err;
5527| }
5528|
5529| //-----
    | -----
5530|
5531| ULONG DeleteExclusionFile (
5532|     HANDLE     *ExclusionFileHandle,
5533|     WCHAR      *ExclusionFileName )
5534| {
5535|     ULONG Err = 0;
5536|     int result = 0;
5537|
5538|     DEBUG_WRITE(L">>> DeleteExclusionFile:
    | filename='%s'\n",ExclusionFileName);
5539|
5540|     if ( *ExclusionFileHandle != NULL ) {
5541|         CloseHandle (*ExclusionFileHandle);
5542|         *ExclusionFileHandle = NULL;
5543|         result = _wremove (ExclusionFileName);
5544|         if ( result != 0 ) {
5545|             int ErrorCode = errno;
5546|             WRITE_ERROR(L"Error: Could not erase
    | exclusion file '%s'\n",ExclusionFileName);
5547|             if ( ErrorCode == EACCES ) {
5548|                 WRITE_ERROR(L"The file is
    | read-only.\n");
5549|             } else if ( ErrorCode == ENOENT ) {
5550|                 WRITE_ERROR(L"The file does not
    | exist.\n");
5551|             } else {
5552|                 WRITE_ERROR(L"Unexpected error
    | %d\n",ErrorCode);
5553|             }
5554|             Err = PSM_ERROR_INVALID_PATH;
5555|         } else {
5556|             DEBUG_WRITE(L">>> Successfully deleted
    | exclusion file '%s'\n",ExclusionFileName);
5557|         }
5558|     } else {
5559|         DEBUG_WRITE(L">>> Warning: called
    | DeleteExclusionFile when file was not open!\n");
5560|     }
5561|
5562|     return Err;
5563| }
5564|
5565| //-----
    | -----

```



```

5566|
5567| ULONG PrepareForVolumeImage (
5568|     plmageCreationInfo OutputImageArray,
5569|     ULONG               OutputImageArrayEntries,
5570|     BOOLEAN             TestOnly,
5571|     ULONG               *NumberOfLocations )
5572| {
5573|     ULONG Err = 0;
5574|     ULONG ThisErr = 0;
5575|     const ULONG OutputImageArrayBytes =
5576|         | sizeof(tlimageCreationInfo) * OutputImageArrayEntries;
5577|     ULONG DataSize=0;
5578|     HKEY Key = INVALID_HANDLE_VALUE;
5579|     WCHAR SubKeyName[64] = {0};
5580|     WCHAR TempDir [256] = {0};
5581|     WCHAR TempPath [256] = {0};
5582|     memset ( OutputImageArray, 0, OutputImageArrayBytes
5583|         | );
5584|     *NumberOfLocations = 0;
5585|     // open the registry
5586|     Err = RegOpenKeyExW(
5587|         HKEY_LOCAL_MACHINE,    // handle of open key
5588|         BackupRegistryPath,    // address of name of
5589|         | subkey to open
5590|         0,                      // reserved
5591|         KEY_READ,              // security access mask
5592|         &Key );                // address of handle of
5593|         | open key
5594|     if ( Err == 0 ) {
5595|         DataSize = sizeof(GloballImageName);
5596|         Err = RegQueryValueExW(
5597|             Key,                // handle of key to
5598|             | query
5599|             L"ImageName",      // address of name
5600|             | of value to query
5601|             NULL,              // reserved
5602|             NULL,              // address of
5603|             | buffer for value type
5604|             (char*)GloballImageName, // address of data
5605|             | buffer
5606|             &DataSize );       // address of data
5607|             | buffer size
5608|     if ( Err == 0 ) {
5609|         int i;
5610|         for ( i=0; i<MAX_LOCATIONS; ++i ) {
5611|             swprintf ( SubKeyName, L"Backup%d", 1+i
5612|                 | );

```

```

5606|
5607|         ThisErr = GetSettingsForBackup (
5608|             Key,
5609|             SubKeyName,
5610|
5611|             | &OutputImageArray[*NumberOfLocations] );
5612|
5613|         if ( ThisErr == 0 ) {
5614|             if (
5615|                 | IsAllWhitespace(OutputImageArray[*NumberOfLocations].Log
5616|                 | onInfo.Path) ) {
5617|                 ThisErr =
5618|                 | PSM_BLANK_ENTRY_IGNORED;
5619|             } else {
5620|                 AppendBackslashIfNeeded (
5621|                 | OutputImageArray[*NumberOfLocations].LogonInfo.Path );
5622|                 ThisErr = RemoteLogonCheck (
5623|                 | &OutputImageArray[*NumberOfLocations].LogonInfo );
5624|                 if ( ThisErr == 0 ) {
5625|                     if ( !TestOnly ) {
5626|                         FILE *ExclusionFile =
5627|                         | 0;
5628|                         ULONG
5629|                         | BackupPathAlreadyExisted = FALSE;
5630|
5631|                         // Check to see if we
5632|                         | can open the exclusion file on what is
5633|                         // currently the first
5634|                         | backup path. (Even though we are about to rename)
5635|                         wscpy ( TempDir,
5636|                         | OutputImageArray[*NumberOfLocations].LogonInfo.Path );
5637|                         AppendBackslashIfNeeded
5638|                         | (TempDir);
5639|                         wscat ( TempDir,
5640|                         | GlobalImageName );
5641|                         wscat ( TempDir,
5642|                         | L".1\\");
5643|
5644|                         BackupPathAlreadyExisted = PathExists(TempDir);
5645|                         if (
5646|                         | !BackupPathAlreadyExisted ) {
5647|                             CreateEntirePath
5648|                             | (TempDir);
5649|                         }
5650|                         ThisErr =
5651|                         | CreateExclusionFile (TempDir, &ExclusionFile,
5652|                         | TempPath);
5653|                         if ( ThisErr == 0 ) {
5654|                             ThisErr =
5655|                             | DeleteExclusionFile (&ExclusionFile, TempPath);

```

```

5636|                if ( ThisErr != 0 )
| {
5637|         | DEBUG_WRITE(L">>> Failed to delete exclusion file
| before rename: %08x\n",ThisErr);
5638|         }
5639|         } else {
5640|         DEBUG_WRITE(L">>>
| Failed to create exclusion file before rename:
| %08x\n",ThisErr);
5641|         }
5642|
5643|         if ( ThisErr == 0 ) {
5644|             if (
| BackupPathAlreadyExisted ) {
5645|             | RenameOldBackupPaths (
5646|             | OutputImageArray[*NumberOfLocations].LogonInfo.Path,
5647|             | GlobalImageName,
5648|             | OutputImageArray[*NumberOfLocations].NumToKeep );
5649|             } else {
5650|             | DEBUG_WRITE(L">>> Backup path did not yet exist - no
| rename of '%s\n",TempDir);
5651|             }
5652|             }
5653|             }
5654|
5655|             if ( ThisErr == 0 ) {
5656|                 wscat (
| OutputImageArray[*NumberOfLocations].LogonInfo.Path,
| GlobalImageName );
5657|                 wscat (
| OutputImageArray[*NumberOfLocations].LogonInfo.Path,
| L".1\\");
5658|                 ThisErr =
| CreateEntirePath (
| OutputImageArray[*NumberOfLocations].LogonInfo.Path );
5659|
5660|                 if ( ThisErr == 0 ) {
5661|                 | OutputImageArray[*NumberOfLocations].LocationNumber =
| 1+i;
5662|                 if ( TestOnly ) {
5663|                 ThisErr =
| TestBackupPath ( &OutputImageArray[*NumberOfLocations]
| );

```

```

5664|                } else {
5665|                // Now that we
| have renamed the paths, create the
5666|                // real
| exclusion file that we will keep open for
5667|                // the whole
| backup...
5668|                ThisErr =
| CreateExclusionFile (
5669|                | OutputImageArray[*NumberOfLocations].LogonInfo.Path,
5670|                | &(OutputImageArray[*NumberOfLocations].ExclusionFileHand
| le),
5671|                | OutputImageArray[*NumberOfLocations].ExclusionFileName
| );
5672|                }
5673|                if ( ThisErr == 0 )
| {
5674|                WRITE(L"Backup
| Path %d = '%s'\n", 1+i,
| OutputImageArray[*NumberOfLocations].LogonInfo.Path);
5675|                | ++(*NumberOfLocations);
5676|                }
5677|                }
5678|                }
5679|                }
5680|                }
5681|                }
5682|
5683|                UpdatePathStatus ( 1+i, ThisErr );
5684|                if ( Err == 0 ) {
5685|                if ( (ThisErr & 0xf0000000) !=
| 0x60000000 ) {
5686|                DEBUG_WRITE (L">>>
| PrepareForVolumeImage: Setting return code to
| %08x\n",ThisErr);
5687|                Err = ThisErr;
5688|                }
5689|                }
5690|                }
5691|                } else {
5692|                WRITE_ERROR(L"Error %08x getting registry
| value 'ImageName' from key
| '%s'\n",Err,BackupRegistryPath);
5693|                }
5694|                } else {
5695|                WRITE_ERROR(L"Error %08x opening registry key

```

```

    | '%s\n",Err,BackupRegistryPath);
5696| }
5697|
5698| return Err;
5699| }
5700|
5701| //-----
    | -----
5702|
5703| ULONG AdjustThreadPriority()
5704| {
5705|     ULONG Err = 0;
5706|     HANDLE hProcess = GetCurrentProcess();
5707|     ULONG PriorityRet = FALSE;
5708|     ULONG PriorityClass = IDLE_PRIORITY_CLASS;
5709|     HKEY Key = INVALID_HANDLE_VALUE;
5710|
5711|     Err = RegOpenKeyExW (
5712|         HKEY_LOCAL_MACHINE,
5713|         BackupRegistryPath,
5714|         0,
5715|         KEY_READ,
5716|         &Key );
5717|
5718|     if ( Err == 0 ) {
5719|         ULONG DataSize = sizeof(PriorityClass);
5720|         Err =
            | RegQueryValueExW(Key,L"ThreadPriority",NULL,NULL,(char*)
            | &PriorityClass,&DataSize);
5721|         if ( Err == 0 ) {
5722|             DEBUG_WRITE(L">>> Got priority value %08x
            | from registry\n",PriorityClass);
5723|         } else {
5724|             DEBUG_WRITE(L">>> Error %08x getting
            | priority from registry - setting to IDLE\n",Err);
5725|             PriorityClass = IDLE_PRIORITY_CLASS;
5726|         }
5727|         RegCloseKey(Key);
5728|         Key = INVALID_HANDLE_VALUE;
5729|     } else {
5730|         DEBUG_WRITE(L">>> Error %08x opening registry
            | key '%s\n",Err,BackupRegistryPath);
5731|     }
5732|
5733|     DEBUG_WRITE(L">>> Current Process Handle =
            | %08x\n",hProcess);
5734|     PriorityRet = SetPriorityClass (hProcess,
            | PriorityClass);
5735|     if ( PriorityRet ) {
5736|         DEBUG_WRITE(L">>> Successfully set priority

```

```

    | class to %08x\n",PriorityClass);
5737| } else {
5738|     Err = GetLastError();
5739|     DEBUG_WRITE(L">>> !!! Error %08x setting
    | priority class to %08x\n",Err,PriorityClass);
5740| }
5741|
5742| return Err;
5743| }
5744|
5745| //-----
    | -----
5746|
5747| void RestoreThreadPriority()
5748| {
5749|     HANDLE hProcess = GetCurrentProcess();
5750|     SetPriorityClass(hProcess, NORMAL_PRIORITY_CLASS);
5751| }
5752|
5753| //-----
    | -----
5754|
5755| ULONG DoVolumeImageBackup (
5756|     const WCHAR *VolumeName,
5757|     const WCHAR *OriginalVolumeName,
5758|     PVOID AbortEvent )
5759| {
5760|     ULONG Err = 0;
5761|     ULONG NumberOfLocations = 0;
5762|     ULONG OutputImageArrayBytes = MAX_LOCATIONS *
    | sizeof(tImageCreationInfo);
5763|     plImageCreationInfo OutputImageArray =
    | (plImageCreationInfo) malloc (OutputImageArrayBytes);
5764|
5765|     DEBUG_WRITE(L">>> DoVolumeImageBackup:
    | VolumeName='%s',
    | AbortEvent=%08x\n",VolumeName,AbortEvent);
5766|
5767|     AdjustThreadPriority();
5768|
5769|     if ( OutputImageArray ) {
5770|         tLocalLogonInfo LogonInfo = {0};
5771|         LocalLogonCheck (&LogonInfo);
5772|
5773|         Err = PrepareForVolumeImage (
5774|             OutputImageArray,
5775|             MAX_LOCATIONS,
5776|             FALSE,
5777|             &NumberOfLocations );
5778|

```

```

5779|     if ( NumberOfLocations > 0 ) {
5780|         Err = BackupManager_CreateVolumeImage (
5781|             &TheBackupManager,
5782|             NumberOfLocations,
5783|             OutputImageArray,
5784|             VolumeName,
5785|             OriginalVolumeName,
5786|             AbortEvent );
5787|     } else {
5788|         WRITE_ERROR(L"Error %08x preparing for
| backup\n",Err);
5789|     }
5790|
5791|     free (OutputImageArray);
5792|     OutputImageArray = NULL;
5793|
5794|     LocalLogoff (&LogonInfo);
5795| } else {
5796|     WRITE_ERROR(L"Error: Out of memory in
| DoVolumeImageBackup!\n");
5797|     Err = ERROR_OUTOFMEMORY;
5798| }
5799|
5800| RestoreThreadPriority();
5801|
5802| return Err;
5803| }
5804|
5805| //-----
| -----
5806|
5807| ULONG BackupManager_ReadyChunkBuffer (
5808|     pBackupManager Manager,
5809|     ULONG      ChunkDataSizeInBytes )
5810| {
5811|     ULONG Status = 0;
5812|     ULONG TotalChunkSizeInBytes =
| sizeof(tVolumeImage_ChunkPrefix) +
| ChunkDataSizeInBytes;
5813|     if ( TotalChunkSizeInBytes >
| Manager->ChunkBufferSize ) {
5814|         pVolumeImage_Chunk NewBuffer =
| (pVolumeImage_Chunk) malloc(TotalChunkSizeInBytes);
5815|         if ( NewBuffer == NULL ) {
5816|             Status = ERROR_OUTOFMEMORY;
5817|         } else {
5818|             if ( (Manager->ChunkBuffer) != NULL ) {
5819|                 free (Manager->ChunkBuffer);
5820|             }
5821|

```

```

5822|         Manager->ChunkBuffer = NewBuffer;
5823|         Manager->ChunkBufferSize =
    | TotalChunkSizeInBytes;
5824|     }
5825| }
5826|
5827| return Status;
5828| }
5829|
5830| //-----
    | -----
5831|
5832| void GenerateChunkChecksums ( pVolumImage_Chunk chunk
    | )
5833| {
5834|     // No more data modifications allowed after
    | ChunkChecksum is calculated!
5835|
5836|     chunk->Prefix.ChunkChecksum = CalculateChecksum (
5837|         chunk->Prefix.ChunkSizeInBytes,
5838|         chunk->Data );
5839|
5840|     // Prefix checksum ***must*** be the last thing
    | modified in the chunk!
5841|     // NOTE: Prefix checksum is checksum of all data
    | in the rest of the prefix,
5842|     // including the chunk checksum. This means
    | changing anything else in the
5843|     // prefix OR the data after the prefix will
    | invalidate the prefix checksum.
5844|
5845|     chunk->Prefix.PrefixChecksum = CalculateChecksum (
5846|         sizeof(tVolumImage_ChunkPrefix) -
    | sizeof(chunk->Prefix.PrefixChecksum),
5847|         &(chunk->Prefix.ChunkChecksum) );
5848| }
5849|
5850| //-----
    | -----
5851|
5852| void AbortBackupStream (
5853|     pImageOutput output,
5854|     ULONG status )
5855| {
5856|     DEBUG_WRITE(L">>> AbortBackupStream called on
    | '%s'\n",output->FileName);
5857|     output->ImageStatus = status;
5858|     CloseImageOutputFile (output);
5859| }
5860|

```



```

5861| //-----
    | -----
5862|
5863| ULONG WriteChunkData (
5864|     plmageOutput      output,
5865|     pVolumImage_Chunk  chunk,
5866|     ULONG              numBytesToWrite )
5867| {
5868|     // NOTE: Assumes that prefix checksums are already
    | valid.
5869|     ULONG Status = 0;
5870|
5871|     if ( output->Handle != INVALID_HANDLE_VALUE ) {
5872|         ULONG NumBytesWritten = 0;
5873|         ULONG DidWrite = WriteFile (
5874|             output->Handle,
5875|             chunk,
5876|             numBytesToWrite,
5877|             &NumBytesWritten,
5878|             NULL );
5879|
5880|         if ( !DidWrite ) {
5881|             Status = GetLastError();
5882|         } else if ( NumBytesWritten != numBytesToWrite
    | ) {
5883|             Status = PSM_ERROR_UNSUCCESSFUL;
5884|         }
5885|
5886|         if ( Status != 0 ) {
5887|             WRITE_ERROR(L"!!! Error %08x writing data
    | to file '%s'\n", Status, output->FileName);
5888|             AbortBackupStream (output, Status);
5889|         }
5890|     }
5891|
5892|     return Status;
5893| }
5894|
5895| //-----
    | -----
5896|
5897| ULONG WriteChunkToSingleFile (
5898|     plmageOutput      output,
5899|     pVolumImage_Chunk chunk,
5900|     ULONG              numBytesToWrite )
5901| {
5902|     const ULONG ONE_MEGABYTE = 1024 * 1024;
5903|     ULONG Status = 0;
5904|     tVolumImage_Chunk ContinuationChunk;
5905|

```

```

5906|    // Before performing the write, make sure we
    | haven't already failed this stream for some reason...
5907|    if ( output->Handle != INVALID_HANDLE_VALUE ) {
5908|        // Update output counters and determine whether
    | we have exceeded size limit...
5909|        output->ByteCounter += numBytesToWrite;
5910|        if ( output->ByteCounter > ONE_MEGABYTE ) {
5911|            output->ByteCounter -= ONE_MEGABYTE;
5912|            ++(output->MegabyteCounter);
5913|            // If MaxSizeInMegabytes is zero, it means
    | that there is no limit to the output file size.
5914|            if ( output->OpenInfo.MaxSizeInMegabytes >
    | 0 ) {
5915|                if ( output->MegabyteCounter >=
    | output->OpenInfo.MaxSizeInMegabytes ) {
5916|                    DEBUG_WRITE(L">>> Reached maximum
    | size of backup image '%s' - continuing in another
    | file.\n",output->FileName);
5917|                    memset ( &ContinuationChunk, 0,
    | sizeof(ContinuationChunk) );
5918|                    ContinuationChunk.Prefix.ChunkType
    | = VICT_END_CONTINUATION;
5919|                    GenerateChunkChecksums (
    | &ContinuationChunk );
5920|                    Status = WriteChunkData ( output,
    | &ContinuationChunk, sizeof(ContinuationChunk.Prefix) );
5921|                    if ( output->Handle !=
    | INVALID_HANDLE_VALUE ) {
5922|                        CloseHandle ( output->Handle );
5923|                        output->Handle =
    | INVALID_HANDLE_VALUE;
5924|                        if ( Status == 0 ) {
5925|                            | ++(output->ContinuationNumber);
5926|                            Status =
    | OpenImageOutputFile (output);
5927|                            if ( Status == 0 ) {
5928|                                memset (
    | &ContinuationChunk, 0, sizeof(ContinuationChunk) );
5929|                                | ContinuationChunk.Prefix.ChunkType =
    | VICT_BEGIN_CONTINUATION;
5930|                                DEBUG_WRITE(L">>>
    | Continuation OpenStatus=%08x\n",Status);
5931|                                GenerateChunkChecksums
    | ( &ContinuationChunk );
5932|                                Status = WriteChunkData
    | ( output, &ContinuationChunk,
    | sizeof(ContinuationChunk.Prefix) );
5933|                                }

```

```

5934|         }
5935|     }
5936| }
5937| }
5938| }
5939|
5940|     if ( Status == 0 ) {
5941|         Status = WriteChunkData ( output, chunk,
5942|             | numBytesToWrite );
5943|     }
5944| }
5945| if ( Status != 0 ) {
5946|     if ( output->ImageStatus == PSM_CREATING_FILES
5947|         | ) {
5948|         DEBUG_WRITE(L">>> WriteChunkToSingleFile:
5949|             | setting '%s' status to
5950|             | %08x\n",output->FileName,Status);
5951|         output->ImageStatus = Status;
5952|     }
5953| }
5954|
5955|
5956| //-----
5957| | -----
5958| ULONG BackupManager_WriteChunk (
5959|     pBackupManager Manager,
5960|     ULONG          ChunkType,
5961|     const ULONG    *UserData,
5962|     ULONG          NumUserData,
5963|     const void     *ChunkData,
5964|     ULONG          ChunkDataSizeInBytes )
5965| {
5966|     ULONG Status = 0;
5967|     ULONG OpenStatus = 0;
5968|     ULONG NumberOfStreamsStillAlive = 0;
5969|
5970|     if ( (NumUserData <= VOLIMAGE_MAX_USER_DATA) &&
5971|         (NumUserData==0 || UserData!=NULL) &&
5972|         (ChunkDataSizeInBytes==0 || ChunkData!=NULL)
5973|         | &&
5974|         Manager->NumImageOutputs>0 ) {
5975|         Status = BackupManager_ReadyChunkBuffer
5976|             | (Manager, ChunkDataSizeInBytes);
5977|
5978|         if ( Status == 0 ) {

```

```

5977|         ULONG i=0;
5978|         BOOLEAN DidWrite=FALSE;
5979|         ULONG NumBytesToWrite =
| sizeof(tVolumeImage_ChunkPrefix) +
| ChunkDataSizeInBytes;
5980|         ULONG NumBytesWritten = 0;
5981|
5982|         memcpy ( Manager->ChunkBuffer->Data,
| ChunkData, ChunkDataSizeInBytes );
5983|         memset ( &Manager->ChunkBuffer->Prefix, 0,
| sizeof(tVolumeImage_ChunkPrefix) );
5984|         Manager->ChunkBuffer->Prefix.ChunkType
| = ChunkType;
5985|
| Manager->ChunkBuffer->Prefix.PrefixSizeInBytes =
| sizeof(tVolumeImage_ChunkPrefix);
5986|
| Manager->ChunkBuffer->Prefix.ChunkSizeInBytes =
| ChunkDataSizeInBytes;
5987|
5988|         for ( i=0; i<NumUserData; ++i ) {
5989|
| Manager->ChunkBuffer->Prefix.UserData[i] = UserData[i];
5990|         }
5991|
5992|
| Manager->ChunkBuffer->Prefix.CumulativeChecksum =
| Manager->CumulativeChecksum;
5993|         Manager->ChunkBuffer->Prefix.ChunkNumber
| = ++(Manager->ChunkCounter);
5994|         GenerateChunkChecksums (
| Manager->ChunkBuffer );
5995|         Manager->CumulativeChecksum ^=
| Manager->ChunkBuffer->Prefix.PrefixChecksum;
5996|
5997|         for ( i=0; i < Manager->NumImageOutputs;
| ++i ) {
5998|             ULONG WriteChunkStatus =
| WriteChunkToSingleFile (
5999|                 &(Manager->ImageOutputArray[i]),
6000|                 Manager->ChunkBuffer,
6001|                 NumBytesToWrite );
6002|
6003|             if ( WriteChunkStatus==0 &&
| Manager->ImageOutputArray[i].Handle!=INVALID_HANDLE_VALU
| E ) {
6004|                 ++NumberOfStreamsStillAlive;
6005|             }
6006|         }
6007|

```

```

6008|         if ( NumberOfStreamsStillAlive == 0 ) {
6009|             if ( Status == 0 ) {
6010|                 Status = PSM_ERROR_UNSUCCESSFUL;
6011|                 WRITE_ERROR(L"Error: All backup
| streams have failed - aborting the backup!\n");
6012|             }
6013|         }
6014|     }
6015| } else {
6016|     Status = PSM_ERROR_INVALID_PARAMETER;
6017| }
6018|
6019| return Status;
6020| }
6021|
6022| //-----
| -----
6023|
6024| ULONG BackupManager_Cleanup ( pBackupManager Manager )
6025| {
6026|     ULONG Status = 0;
6027|
6028|     if ( Manager->ChunkBuffer != NULL ) {
6029|         free (Manager->ChunkBuffer);
6030|         Manager->ChunkBuffer = NULL;
6031|     }
6032|
6033|     Manager->ChunkBufferSize = 0;
6034|     LogonShareCleanup();
6035|
6036|     return Status;
6037| }
6038|
6039| //-----
| -----
6040|
6041| BOOLEAN AllBytesSame (
6042|     pPSM_VIBM_Granule Granule )
6043| {
6044|     int numBytes = Granule->GranuleSizeInBytes;
6045|     if ( numBytes >= 2 ) {
6046|         const unsigned char *data = (const unsigned
| char *) (Granule->Data);
6047|         int i;
6048|         for ( i=1; i < numBytes; ++i ) {
6049|             if ( data[i] != data[0] ) {
6050|                 return FALSE;
6051|             }
6052|         }
6053|     }

```

```

6054|     return TRUE; // Crunch time
6055| }
6056|
6057|     return FALSE; // There's no need to compress less
        | than 2 bytes, now is there?
6058| }
6059|
6060| //-----
        | -----
6061|
6062| ULONG BackupManager_ProcessGranules (
6063|     pBackupManager      Manager,
6064|     pPSM_VIBM_Granule    Granule )
6065| {
6066|     ULONG Status = 0;
6067|     ULONG UserData[5] = {
6068|         Granule->GranuleOffsetInBytes.LowPart,    //
        | [0]
6069|         Granule->GranuleOffsetInBytes.HighPart,    //
        | [1]
6070|         VI_COMPRESS_NONE,                        //
        | [2]
6071|         0,                                        //
        | [3]
6072|         0                                        //
        | [4]
6073|     };
6074|
6075|     const void *DataToWrite = Granule->Data;
6076|     ULONG BytesToWrite = Granule->GranuleSizeInBytes;
6077|
6078|     if ( AllBytesSame(Granule) ) {
6079|         UserData[2] = VI_COMPRESS_ALL_BYTES_SAME;
6080|         UserData[3] = (ULONG) ( *((unsigned char *)
        | (Granule->Data)) );
6081|         UserData[4] = Granule->GranuleSizeInBytes;
6082|         BytesToWrite = 0;
6083|         DataToWrite = NULL;
6084|     }
6085|
6086|     Status = BackupManager_WriteChunk (
6087|         Manager,
6088|         VICT_GRANULE,
6089|         UserData,
6090|         sizeof(UserData) / sizeof(UserData[0]),
6091|         DataToWrite,
6092|         BytesToWrite );
6093|
6094|     if(Manager->BackupAbortEvent){
6095|

```

```

    | if(WaitForSingleObject(Manager->BackupAbortEvent,0)==WAI
    | T_OBJECT_0) {
6096|         Status = ERROR_REQUEST_ABORTED;
6097|     }
6098| }
6099|
6100| return Status;
6101| }
6102|
6103|
6104| //-----
    | -----
6105|
6106| DWORD SS_VolumeImageCallBack (
    | pPSM_VolumeImageCallBackParms parms )
6107| {
6108|     DWORD Status = 0; // returning any nonzero value
    | will abort volume image operation
6109|
6110|     switch ( parms->MessageType ) {
6111|         case PSM_VIBMT_START: {
6112|             pPSM_VIBMT_Start Start =
    | &parms->MessageData.Start;
6113|             ULONG UserData[] = {
6114|                 Start->SizeInBytes.LowPart,
6115|                 Start->SizeInBytes.HighPart,
6116|                 Start->NumUsedClusters.LowPart,
6117|                 Start->NumUsedClusters.HighPart,
6118|                 Start->ClusterSize };
6119|
6120|             TheBackupManager.CumulativeChecksum = 0;
6121|             TheBackupManager.ChunkCounter = 0;
6122|
6123|             Status = BackupManager_WriteChunk (
6124|                 &TheBackupManager,
6125|                 VICT_START,
6126|                 UserData,
6127|                 sizeof(UserData) / sizeof(UserData[0]),
6128|                 NULL,
6129|                 0 );
6130|         } break;
6131|
6132|         case PSM_VIBMT_PARTITION_INFO: {
6133|             pPSM_VIBMT_PartitionInfo PartitionInfo =
    | &parms->MessageData.PartitionInfo;
6134|             DEBUG_WRITE(L">>> PartitionInfo:\n" );
6135|             DEBUG_WRITE(L">>>   Volume Serial Number
    | = %08x\n", PartitionInfo->VolumeSerialNumber );
6136|             DEBUG_WRITE(L">>>   Volume Unique Id
    | = %08x\n", PartitionInfo->VolumeUniqueId );

```

```

6137|         DEBUG_WRITE(L">>> Partition Length
| = %016l64x\n", PartitionInfo->PartitionLength.QuadPart
| );
6138|         Status = BackupManager_WriteChunk (
6139|             &TheBackupManager,
6140|             VICT_PARTITION_INFO,
6141|             NULL,
6142|             0,
6143|             PartitionInfo,
6144|             sizeof(*PartitionInfo) );
6145|     } break;
6146|
6147|     case PSM_VIBMT_GRANULE: {
6148|         pPSM_VIBM_Granule Granule =
| &parms->MessageData.Granule;
6149|         Status = BackupManager_ProcessGranules
| (&TheBackupManager, Granule);
6150|     } break;
6151|
6152|     case PSM_VIBMT_FINISH: {
6153|         Status = BackupManager_WriteChunk (
| &TheBackupManager, VICT_FINISH, 0, 0, NULL, 0 );
6154|     } break;
6155| }
6156|
6157| return Status;
6158| }
6159|
6160| //-----
| -----
6161|
6162| ULONG OpenImageOutputFile ( plImageOutput FileInfo )
6163| {
6164|     ULONG Err = 0;
6165|
6166|     FileInfo->ByteCounter =
| sizeof(tVolumelImage_ChunkPrefix); // allow room for
| continuation chunk at end
6167|     FileInfo->MegabyteCounter = 0;
6168|     FileInfo->ImageStatus = STATUS_PENDING;
6169|
6170|     swprintf ( FileInfo->FileName, L"%simage.%03d",
6171|         FileInfo->OpenInfo.LogonInfo.Path,
6172|         FileInfo->ContinuationNumber );
6173|
6174|     FileInfo->Handle = CreateFileW(
6175|         FileInfo->FileName, // file
| name
6176|         GENERIC_READ | GENERIC_WRITE, // access
| mode

```



```

6177|    0,                // share
    | mode
6178|    NULL,             // SD
6179|    CREATE_ALWAYS,    // how to
    | create
6180|    FILE_ATTRIBUTE_NORMAL,    // file
    | attributes
6181|    NULL );          // handle
    | to template file
6182|
6183|    if ( FileInfo->Handle == INVALID_HANDLE_VALUE ) {
6184|        FileInfo->ImageStatus = Err = GetLastError();
6185|        UpdatePathStatus (
    | FileInfo->OpenInfo.LocationNumber,
    | PSM_ERROR_INVALID_PATH );
6186|        WRITE_ERROR(L"Error %08x trying to open image
    | output file '%s'\n",Err,FileInfo->FileName);
6187|    } else {
6188|        DEBUG_WRITE(L">>> OpenImageOutputFile:
    | Successfully created '%s'\n",FileInfo->FileName);
6189|        UpdatePathStatus (
    | FileInfo->OpenInfo.LocationNumber,
    | PSM_BACKING_UP_VOLUME );
6190|        FileInfo->ImageStatus = PSM_CREATING_FILES;
6191|    }
6192|
6193|    return Err;
6194| }
6195|
6196| //-----
    | -----
6197|
6198| void EraseOldSummaryFile ( pImageOutput FileInfo )
6199| {
6200|     WCHAR SummaryFileName [256];
6201|     int result = 0;
6202|
6203|     // delete summary file...
6204|     wcscpy ( SummaryFileName,
    | FileInfo->OpenInfo.LogonInfo.Path );
6205|     AppendBackslashIfNeeded ( SummaryFileName );
6206|     wcscat ( SummaryFileName, L"image.000" );
6207|     result = _wremove ( SummaryFileName );
6208|     DEBUG_WRITE(L">>> EraseOldSummaryFile:
    | SummaryFileName='%s', result=%d,
    | errno=%d\n",SummaryFileName,result,errno);
6209|
6210|     // delete html log file...
6211|     wcscpy ( SummaryFileName,
    | FileInfo->OpenInfo.LogonInfo.Path );

```

```

6212| AppendBackslashIfNeeded ( SummaryFileName );
6213| wscat ( SummaryFileName, L"image.htm" );
6214| result = _wremove ( SummaryFileName );
6215| DEBUG_WRITE(L">>> EraseOldSummaryFile:
    | HtmlFileName='%s', result=%d,
    | errno=%d\n",SummaryFileName,result,errno);
6216| }
6217|
6218| //-----
    | -----
6219|
6220| ULONG GenerateSummaryFile (
6221|     pImageOutput FileInfo,
6222|     ULONG WarningCode )
6223| {
6224|     ULONG Err = 0;
6225|     WCHAR FileName [256];
6226|     WCHAR *RegPath = FileName;    // scary aliasing!
    | can't use RegPath and FileName at same time.
6227|     FILE *SummaryFile = NULL;
6228|     ULONG BackupNumber = 0;
6229|     ULONG BackupCount = 0;
6230|     ULONG TimeStampErr = 0;
6231|     WCHAR TimeStamp [64];
6232|     WCHAR Identity [256];
6233|
6234|     wcscpy ( FileName,
    | FileInfo->OpenInfo.LogonInfo.Path );
6235|     AppendBackslashIfNeeded ( FileName );
6236|     wscat ( FileName, L"image.000" );
6237|
6238|     DEBUG_WRITE(L">>> GenerateSummaryFile:
    | FileName='%s',
    | WarningCode=%08x\n",FileName,WarningCode);
6239|     SummaryFile = _wfopen(FileName,L"wt");
6240|     if ( SummaryFile == NULL ) {
6241|         WRITE_ERROR(L"Error: Could not open summary
    | file '%s' for write!\n",FileName);
6242|         Err = PSM_ERROR_INVALID_PATH;
6243|     } else {
6244|         ULONG NameLength = 0;
6245|
6246|         wcscpy ( Identity,
    | FileInfo->OpenInfo.LogonInfo.Path );
6247|         NameLength = wcslen(Identity);
6248|         if ( NameLength > 1 ) {
6249|             // The path will be something like
    | 'd:\vimage.psm\backup.1\'.
6250|             // Truncate at the final '.' to get
    | 'd:\vimage.psm\backup'

```

```

6251|
6252|     ULONG NamePos = NameLength-1;
6253|     if ( Identity[NamePos] == '\\' ) {
6254|         --NamePos;
6255|     }
6256|
6257|     while ( NamePos > 0 ) {
6258|         if ( Identity[NamePos] == '.' ) {
6259|             DEBUG_WRITE(L">>>
| GenerateSummaryFile: Before truncate:
| Identity='%s\n",Identity);
6260|             Identity[NamePos] = '\0';
6261|             DEBUG_WRITE(L">>>
| GenerateSummaryFile: After truncate:
| Identity='%s\n",Identity);
6262|             break;
6263|         } else if ( Identity[NamePos] == '\\' )
| {
6264|             // Looks like there wasn't a '.'
| where there was supposed to be.
6265|             // Oh well...
6266|             break;
6267|         }
6268|         --NamePos;
6269|     }
6270| }
6271|
6272|     fprintf
| (SummaryFile,"NumFilesInBackup=%d\n",FileInfo->Continuat
| ionNumber);
6273|     fprintf
| (SummaryFile,"WarningCode=%d\n",WarningCode);
6274|     fprintf (SummaryFile,"Identity=%S\n",Identity);
6275|     fclose(SummaryFile);
6276|     SummaryFile = NULL;
6277|
6278|     // Now update "CurrentCopies" in the registry.
6279|     // Do this by counting up the number of
| "image.000" files that are openable...
6280|     BackupCount = 0;
6281|     for ( BackupNumber=1; BackupNumber <=
| MAX_IMAGES_PER_LOCATION; ++BackupNumber ) {
6282|         wcsncpy ( FileName,
| FileInfo->OpenInfo.LogonInfo.Path ); //
| "d:\blahblah\blah\fred.3[]"
6283|         AppendBackslashIfNeeded ( FileName );
| // "d:\blahblah\blah\fred.3\"
6284|         NameLength = wcslen(FileName);
6285|         if ( NameLength >= 2 ) {
6286|             FileName[NameLength-2] = (WCHAR)

```

```

    | (BackupNumber + '0');
6287|         wscat(FileName,L"image.000");
6288|         DEBUG_WRITE(L">>> CurrentCopies check:
    | FileName='%s'\n",FileName);
6289|         SummaryFile = _wopen(FileName,L"rt");
6290|         if ( SummaryFile != NULL ) {
6291|             fclose(SummaryFile);
6292|             SummaryFile = NULL;
6293|             ++BackupCount;
6294|         }
6295|     }
6296| }
6297|
6298|     DEBUG_WRITE(L">>> CurrentCopies check:
    | BackupCount=%d\n", BackupCount);
6299|     UpdateBackupCount (
    | FileInfo->OpenInfo.LocationNumber, BackupCount );
6300|
6301|     TimeStampErr = GetTimeStamp(TimeStamp);
6302|     if ( TimeStampErr == 0 ) {
6303|         swprintf ( RegPath, L"%sBackup%d",
    | BackupRegistryPath, FileInfo->OpenInfo.LocationNumber
    | );
6304|         UpdateTimeStamp ( RegPath, L"LastBackup",
    | TimeStamp );
6305|     } else {
6306|         WRITE_ERROR(L"Warning: Could not update
    | LastBackup time stamp in registry for location #%%d
    | (error
    | %%08x)\n",FileInfo->OpenInfo.LocationNumber,TimeStampErr)
    | ;
6307|     }
6308| }
6309|
6310| return Err;
6311| }
6312|
6313| //-----
    | -----
6314|
6315| ULONG CloseImageOutputFile ( plmageOutput FileInfo )
6316| {
6317|     ULONG Err = 0;
6318|
6319|     if ( FileInfo->OpenInfo.ExclusionFileHandle!=NULL
    | ) {
6320|         Err = DeleteExclusionFile (
6321|             &(FileInfo->OpenInfo.ExclusionFileHandle),
6322|             FileInfo->OpenInfo.ExclusionFileName );
6323|     }

```

```

6324|
6325|  if ( FileInfo->Handle != NULL && FileInfo->Handle
    | != INVALID_HANDLE_VALUE ) {
6326|      CloseHandle (FileInfo->Handle);
6327|      FileInfo->Handle = INVALID_HANDLE_VALUE;
6328|      if ( FileInfo->ImageStatus ==
    | PSM_CREATING_FILES ) {
6329|          FileInfo->ImageStatus = STATUS_SUCCESS;
6330|      } else {
6331|          DeleteAllFilesAndDirectory (
    | FileInfo->OpenInfo.LogonInfo.Path );
6332|      }
6333|      UpdateLastBackupResult (
    | FileInfo->OpenInfo.LocationNumber,
    | FileInfo->ImageStatus );
6334|      UpdatePathStatus (
    | FileInfo->OpenInfo.LocationNumber, PSM_VALID_PATH );
6335|  }
6336|
6337|  RemoteLogoff ( &(FileInfo->OpenInfo.LogonInfo) );
6338|  return Err;
6339| }
6340|
6341| //-----
    | -----
6342|
6343| void BackupManager_SetAllErrorCodes (
6344|     pBackupManager Manager,
6345|     ULONG          Error )
6346| {
6347|     // This function is called when an error is
    | encountered that affects
6348|     // all backup streams, such as the backup being
    | aborted.
6349|     // We set all the error codes so that the backups
    | will be deleted
6350|     // after the files are closed.
6351|
6352|     ULONG i;
6353|     for ( i=0; i < Manager->NumImageOutputs; ++i ) {
6354|         // Only set error codes for streams that
    | currently think they are successful...
6355|         if ( Manager->ImageOutputArray[i].ImageStatus
    | == PSM_CREATING_FILES ) {
6356|             DEBUG_WRITE(L">>> Setting ImageStatus for
    | '%s' to
    | %08x\n",Manager->ImageOutputArray[i].FileName,Error);
6357|             Manager->ImageOutputArray[i].ImageStatus =
    | Error;
6358|         }

```

```

6359|  }
6360| }
6361|
6362| //-----
    | -----
6363|
6364| ULONG BackupManager_CreateVolumeImage (
6365|     pBackupManager    Manager,
6366|     int                NumOutputFiles,
6367|     plmageCreationInfo OutputArray,
6368|     const WCHAR        *_VolName,
6369|     const WCHAR        *_OriginalVolumeName,
6370|     PVOID              AbortEvent )
6371| {
6372|     ULONG Err = 0;
6373|     WCHAR *OriginalVolumeName = NULL;
6374|
6375|     DEBUG_WRITE(L">>> CreateVolumeImage:
    | NumOutputFiles=%d, _VolName='%s',
    | _Original='%s'\n", NumOutputFiles, _VolName, _OriginalVolum
    | eName);
6376|
6377|     Manager->BackupAbortEvent = AbortEvent;
6378|
6379|     if ( NumOutputFiles<1 ||
        | NumOutputFiles>MAX_LOCATIONS ) {
6380|         Err = PSM_ERROR_INVALID_PARAMETER;
6381|         WRITE_ERROR(L"Error: Invalid number of image
        | output files: %d\n", NumOutputFiles);
6382|     } else {
6383|         int i;
6384|         for ( i=0; i<NumOutputFiles; ++i ) {
6385|             Manager->ImageOutputArray[i].OpenInfo =
        | OutputArray[i];
6386|
        | Manager->ImageOutputArray[i].ContinuationNumber = 1;
6387|             Err = OpenImageOutputFile (
        | &Manager->ImageOutputArray[i] );
6388|
        | UpdateLastBackupResult (
6390|         | Manager->ImageOutputArray[i].OpenInfo.LocationNumber,
6391|         | Manager->ImageOutputArray[i].ImageStatus );
6392|
6393|         if ( Err == 0 ) {
6394|             EraseOldSummaryFile (
        | &Manager->ImageOutputArray[i] );
6395|         } else {
6396|             while ( --i >= 0 ) {

```

```

6397|         CloseImageOutputFile (
6398|             | &Manager->ImageOutputArray[i] );
6399|         }
6400|         break;
6401|     }
6402| }
6403|
6404| if ( Err == 0 ) {
6405|     Manager->NumImageOutputs = NumOutputFiles;
6406|     if ( _VolName ) {
6407|         int VolNameLength = wcslen(_VolName);
6408|         WCHAR *VolName = (WCHAR *)
6409|             | LocalAlloc(LPTR,sizeof(WCHAR)*(2+VolNameLength));
6410|         if ( VolName ) {
6411|             const unsigned VOLUME_GUID_CHARS =
6412|                 | 256;
6413|             const unsigned VOLUME_GUID_BYTES =
6414|                 | sizeof(WCHAR) * VOLUME_GUID_CHARS;
6415|             WCHAR *VolumeGuid =
6416|                 | LocalAlloc(LPTR,VOLUME_GUID_BYTES);
6417|             wcsncpy ( VolName, _VolName );
6418|             if ( VolNameLength>0 &&
6419|                 | VolName[VolNameLength-1]!='\\' ) {
6420|                 VolName[VolNameLength++] =
6421|                     | "\\";
6422|                 VolName[VolNameLength] = 0;
6423|             }
6424|             if ( VolumeGuid ) {
6425|                 BOOL GotName =
6426|                     | GetVolumeNameForVolumeMountPointW (
6427|                         VolName,
6428|                         VolumeGuid,
6429|                         VOLUME_GUID_BYTES );
6430|                 if ( GotName ) {
6431|                     // Fix Volume Guids... need
6432|                     | final backslash to be ABSENT!
6433|                     int VolumeGuidLength =
6434|                         | wcslen(VolumeGuid);
6435|                     if ( VolumeGuidLength>0 &&
6436|                         | VolumeGuid[VolumeGuidLength-1]=='\\' ) {
6437|                         VolumeGuid[VolumeGuidLength-1] = 0;
6438|                     }
6439|                     WRITE(L""%s' = '%s\n",
6440|                         | VolName, VolumeGuid);
6441|                 }
6442|             }
6443|         }

```

```

6434|             OriginalVolumeName = (WCHAR
| *)LocalAlloc(LPTR,sizeof(WCHAR)*(2+wcslen(_OriginalVolum
| eName)));
6435|             if ( OriginalVolumeName ) {
6436|                 | wcsncpy(OriginalVolumeName, _OriginalVolumeName);
6437|                 AppendBackslashIfNeeded
| (OriginalVolumeName);
6438|                 UpdateEngineStatus (
| PSM_BACKING_UP_VOLUME );
6439|                 Err =
| Psm_VolumeImageDumpW ( VolumeGuid, OriginalVolumeName,
| SS_VolumeImageCallback );
6440|                 UpdateEngineStatus (
| PSM_IDLE );
6441|
6442|                 if ( Err != 0 ) {
6443|                     | BackupManager_SetAllErrorCodes (Manager, Err); // kill
| backups on close (later)
6444|                     | WRITE_ERROR(L"Error: Volume Image Dump returned
| %08x\n",Err);
6445|                     WRITE_ERROR(L"The
| backup has been canceled.\n");
6446|                 }
6447|                 | LocalFree(OriginalVolumeName);
6448|                 OriginalVolumeName = 0;
6449|             } else {
6450|                 Err =
| ERROR_OUTOFMEMORY;
6451|                 WRITE_ERROR(L"Error:
| Out of memory (OriginalVolumeName)\n");
6452|             }
6453|             } else {
6454|                 Err = GetLastError();
6455|                 WRITE_ERROR(L"Could not get
| volume guid for '%s'; error=%08x\n", VolName, Err );
6456|             }
6457|
6458|             LocalFree(VolumeGuid);
6459|             VolumeGuid = NULL;
6460|         } else {
6461|             WRITE_ERROR(L"Out of memory
| (VolumeGuids) in CreateVolumeImageFile()\n");
6462|             Err = ERROR_OUTOFMEMORY;
6463|         }
6464|     } else {
6465|         WRITE_ERROR(L"Out of memory

```



```

    | (VolumeNames) in CreateVolumeImageFile()\n");
6466|         Err = ERROR_OUTOFMEMORY;
6467|     }
6468|
6469|     if ( VolName ) {
6470|         LocalFree(VolName);
6471|         VolName = NULL;
6472|     }
6473|     } else {
6474|         WRITE_ERROR(L"Internal error: invalid
    | parameter to CreateVolumeImageFile()\n");
6475|         Err = PSM_ERROR_INVALID_PARAMETER;
6476|     }
6477|
6478|     for ( i=0; i<NumOutputFiles; ++i ) {
6479|         if (
    | Manager->ImageOutputArray[i].Handle !=
    | INVALID_HANDLE_VALUE ) {
6480|             if (
    | Manager->ImageOutputArray[i].ImageStatus ==
    | PSM_CREATING_FILES ) {
6481|                 GenerateSummaryFile (
    | &Manager->ImageOutputArray[i], Manager->NonFatalError
    | );
6482|             }
6483|             CloseImageOutputFile (
    | &Manager->ImageOutputArray[i] );
6484|         }
6485|
6486|         UpdateLastBackupResult (
6487|             | Manager->ImageOutputArray[i].OpenInfo.LocationNumber,
6488|             | Manager->ImageOutputArray[i].ImageStatus );
6489|     }
6490|     Manager->NumImageOutputs = 0;
6491|     BackupManager_Cleanup ( Manager );
6492| }
6493| }
6494|
6495| Manager->BackupAbortEvent = NULL;
6496|
6497| if ( Err == 0 ) {
6498|     if ( Manager->NonFatalError != 0 ) {
6499|         Err = Manager->NonFatalError;
6500|     }
6501| }
6502|
6503| Manager->NonFatalError = 0;
6504|

```

```

6505|  DEBUG_WRITE(L">>> CreateVolumeImage returning
      | %08x\n",Err);
6506|  return Err;
6507| }
6508|
6509| //-----
      | -----
6510|
6511| ULONG CalculateChecksum (
6512|  ULONG    DataSizeInBytes,
6513|  const void *DataBuffer )
6514| {
6515|  ULONG sum = 0xc9d4b5a2;
6516|  ULONG multiplier = 0x1a2b3c4d;
6517|  const unsigned char *p = (const unsigned char *)
      | DataBuffer;
6518|  ULONG i;
6519|
6520|  for ( i=0; i < DataSizeInBytes; ++i ) {
6521|      sum ^= (0x100 + *p++) * multiplier++;
6522|      sum = (sum << 9) | (sum >> (32-9));  //
      | rotate left 9 bits
6523|  }
6524|
6525|  if ( sum == CHECKSUM_IGNORE_DWORD ) {
6526|      sum = 0xffffffff; // not allowed to be
      | CHECKSUM_IGNORE_DWORD
6527|  }
6528|
6529|  return sum;
6530| }
6531|
6532| //-----
      | -----
6533|
6534| BOOLEAN ValidateChecksum (
6535|  ULONG    DataSizeInBytes,
6536|  const void *DataBuffer,
6537|  ULONG    StoredChecksum )
6538| {
6539|  BOOLEAN isValid = TRUE;
6540|
6541|  // If the stored checksum is CHECKSUM_IGNORE_DWORD,
      | it means we should ignore it.
6542|
6543|  if ( StoredChecksum != CHECKSUM_IGNORE_DWORD ) {
6544|      ULONG CalcChecksum = CalculateChecksum
      | (DataSizeInBytes, DataBuffer);
6545|      isValid = (CalcChecksum == StoredChecksum);
6546|  }

```

```

6547|
6548|     return isValid;
6549| }
6550|
6551| //-----
    | -----
6552|
6553| ULONG UpdateBackupStatus (
6554|     const WCHAR * const RegPath,
6555|     const WCHAR * const ValueName,
6556|     ULONG Status )
6557| {
6558|     HKEY Key = INVALID_HANDLE_VALUE;
6559|     ULONG Err = RegOpenKeyExW(
6560|         HKEY_LOCAL_MACHINE,    // handle of open key
6561|         RegPath,                // address of name of
        | subkey to open
6562|         0,                      // reserved
6563|         KEY_ALL_ACCESS,        // security access mask
6564|         &Key );                // address of handle of
        | open key
6565|
6566|     if ( Err == 0 ) {
6567|         Err = RegSetValueExW (
6568|             Key,
6569|             ValueName,
6570|             0,
6571|             REG_DWORD,
6572|             (const unsigned char *)&Status,
6573|             sizeof(ULONG) );
6574|
6575|         if ( Err != 0 ) {
6576|             WRITE_ERROR(L"UpdateBackupStatus: Could
        | not write value '%s' to key '%s', error
        | %08x\n",ValueName,RegPath,Err);
6577|         } else {
6578|             DEBUG_WRITE(L">>> Just wrote %08x to value
        | '%s' in key '%s'\n",Status,ValueName,RegPath);
6579|         }
6580|
6581|         RegCloseKey(Key);
6582|         Key = INVALID_HANDLE_VALUE;
6583|     } else {
6584|         WRITE_ERROR(L"UpdateBackupStatus: Could not
        | open registry key '%s', error %08x\n",RegPath,Err);
6585|     }
6586|
6587|     return Err;
6588| }
6589|

```

```

6590| //-----
    | -----
6591|
6592| ULONG UpdateTimeStamp (
6593|     const WCHAR * const RegPath,
6594|     const WCHAR * const ValueName,
6595|     const WCHAR * const TimeStamp )
6596| {
6597|     ULONG Err=0;
6598|     DWORD len=0;
6599|     HKEY Key = INVALID_HANDLE_VALUE;
6600|
6601|     len = wcslen(TimeStamp);
6602|     len = (len*sizeof(WCHAR))+(sizeof(WCHAR)*2);
6603|
6604|     Err = RegOpenKeyExW(
6605|         HKEY_LOCAL_MACHINE,    // handle of open key
6606|         RegPath,              // address of name of
        | subkey to open
6607|         0,                    // reserved
6608|         KEY_ALL_ACCESS,       // security access mask
6609|         &Key );               // address of handle of
        | open key
6610|
6611|     if ( Err == 0 ) {
6612|         Err = RegSetValueExW (
6613|             Key,
6614|             ValueName,
6615|             0,
6616|             REG_SZ,
6617|             (const unsigned char *)TimeStamp,
6618|             len );
6619|
6620|         if ( Err != 0 ) {
6621|             WRITE_ERROR(L"UpdateBackupResult: Could not
        | write value '%s' to key '%s', error
        | %08x\n",ValueName,RegPath,Err);
6622|         } else {
6623|             DEBUG_WRITE(L">>> Just wrote %s to value
        | '%s' in key '%s\n",TimeStamp,ValueName,RegPath);
6624|         }
6625|
6626|         RegCloseKey(Key);
6627|         Key = INVALID_HANDLE_VALUE;
6628|     } else {
6629|         WRITE_ERROR(L"UpdateBackupResult: Could not
        | open registry key '%s', error %08x\n",RegPath,Err);
6630|     }
6631|
6632|     return Err;

```

```

6633| }
6634|
6635| //-----
| -----
6636|
6637| ULONG ConvertErrorCodeToRegistryStatus ( ULONG
| ErrorCode )
6638| {
6639|     // This function converts an error code into a code
| that
6640|     // the WebUI can display as a brief string.
6641|
6642|     ULONG StatusCode = PSM_ERROR_UNSUCCESSFUL;
6643|     if ( ErrorCode & 0x20000000 ) {
6644|         // If the customer bit is set, assume it's a
| valid message.
6645|         StatusCode = ErrorCode;
6646|     } else {
6647|         // Figure out what the error is and translate
| it if possible...
6648|         switch ( ErrorCode ) {
6649|             case STATUS_SUCCESS:         StatusCode
| = PSM_OPERATION_SUCCESSFUL;     break;
6650|             case ERROR_REQUEST_ABORTED:   StatusCode
| = PSM_CANCELED_BY_USER;         break;
6651|             case ERROR_OUTOFMEMORY:       StatusCode
| = PSM_ERROR_OUT_OF_MEMORY;      break;
6652|         }
6653|     }
6654|
6655|     DEBUG_WRITE(L">>> ConvertErrorCodeToRegistryStatus:
| %08x translated into %08x\n",ErrorCode,StatusCode);
6656|     return StatusCode;
6657| }
6658|
6659| //-----
| -----
6660|
6661| ULONG UpdateEngineStatus ( ULONG Status )
6662| {
6663|     ULONG Err = 0;
6664|     DEBUG_WRITE(L">>> UpdateEngineStatus:
| Status=%08x\n", Status);
6665|     Status = ConvertErrorCodeToRegistryStatus (Status);
6666|     Err = UpdateBackupStatus ( BackupRegistryPath,
| L"EngineStatus", Status );
6667|     return Err;
6668| }
6669|
6670| //-----

```

```

| -----
6671|
6672| ULONG UpdateLastBackupResult ( int LocationNumber,
| ULONG Status )
6673| {
6674|     WCHAR RegPath [256];
6675|     ULONG Err = 0;
6676|
6677|     DEBUG_WRITE(L">>> UpdateLastBackupResult:
| LocationNumber=%d,
| Status=%08x\n",LocationNumber,Status);
6678|     Status = ConvertErrorCodeToRegistryStatus (Status);
6679|     swprintf ( RegPath, L"%sBackup%d\\",
| BackupRegistryPath, LocationNumber );
6680|     Err = UpdateBackupStatus ( RegPath,
| L"LastBackupResult", Status );
6681|
6682|     return Err;
6683| }
6684|
6685| //-----
| -----
6686|
6687| ULONG UpdatePathStatus ( int LocationNumber, ULONG
| Status )
6688| {
6689|     WCHAR RegPath [256];
6690|     ULONG Err = 0;
6691|
6692|     if ( !(Status & 0x20000000) ) {
6693|         // customer bit not set... need to translate!
6694|         if ( Status == STATUS_SUCCESS ) {
6695|             Status = PSM_VALID_PATH;
6696|         } else {
6697|             DEBUG_WRITE(L">>> UpdatePathStatus:
| translating error %08x into
| PSM_ERROR_INVALID_PATH\n",Status);
6698|             Status = PSM_ERROR_INVALID_PATH;
6699|         }
6700|     }
6701|
6702|     DEBUG_WRITE(L">>> UpdatePathStatus:
| LocationNumber=%d,
| Status=%08x\n",LocationNumber,Status);
6703|     swprintf ( RegPath, L"%sBackup%d\\",
| BackupRegistryPath, LocationNumber );
6704|     Err = UpdateBackupStatus ( RegPath, L"PathStatus",
| Status );
6705|
6706|     return Err;

```

```

6707| }
6708|
6709| //-----
    | -----
6710|
6711| ULONG UpdateBackupCount ( int LocationNumber, ULONG
    | BackupCount )
6712| {
6713|     WCHAR RegPath [256];
6714|     ULONG Err = 0;
6715|
6716|     DEBUG_WRITE(L">>> UpdateBackupCount:
    | LocationNumber=%d,
    | Count=%d\n",LocationNumber,BackupCount);
6717|     swprintf ( RegPath, L"%sBackup%d\\",
    | BackupRegistryPath, LocationNumber );
6718|     Err = UpdateBackupStatus ( RegPath,
    | L"CurrentCopies", BackupCount );
6719|
6720|     return Err;
6721| }
6722|
6723| //-----
    | -----
6724|
6725| ULONG ChunkFileIsCorrupt (
6726|     const WCHAR *FileName,
6727|     FILE      *BackupFile,
6728|     int       ContinuationNumber,
6729|     int       TotalNumberOfFiles,
6730|     ULONG     *CumulativeChecksum,
6731|     ULONG     *ChunkCounter )
6732| {
6733|     tVolumeImage_ChunkPrefix prefix = {0};
6734|     int ChunkNumber = 0;    // chunk number in this
    | file only (not to be confused with ChunkCounter for all
    | files)
6735|     BOOLEAN ValidChecksum = FALSE;
6736|     ULONG Err = 0;
6737|     const ULONG MaxDataBytes = 128 * 1024;
6738|     char *DataBuffer = (char *)malloc (MaxDataBytes);
6739|     BOOLEAN FoundFinishChunk = FALSE;
6740|     BOOLEAN FoundEndContinuationChunk = FALSE;
6741|
6742|     DEBUG_WRITE(L">>> Checking file '%s'\n", FileName);
6743|
6744|     if ( DataBuffer ) {
6745|         while ( Err==0 &&
    | fread(&prefix,sizeof(prefix),1,BackupFile)==1 ) {
6746|             ++ChunkNumber;

```

```

6747|         ValidChecksum = ValidateChecksum (
6748|             sizeof(prefix) -
        | sizeof(prefix.PrefixChecksum),
6749|             ((const char *)&prefix) +
        | sizeof(prefix.PrefixChecksum),
6750|             prefix.PrefixChecksum );
6751|
6752|         if ( !ValidChecksum ) {
6753|             Err = PSM_CORRUPT_BACKUP;
6754|             WRITE_ERROR(L"Error: Prefix checksum
        | failure in '%s' - file is corrupt (chunk number =
        | %d)\n",FileName,ChunkNumber);
6755|         } else if ( prefix.ChunkSizeInBytes >
        | MaxDataBytes ) {
6756|             Err = PSM_CORRUPT_BACKUP;
6757|             WRITE_ERROR(L"Error: Data size invalid
        | in '%s' - file is corrupt\n",FileName);
6758|         } else {
6759|             if ( prefix.ChunkSizeInBytes > 0 ) {
6760|                 ULONG NumBytesRead =
        | fread(DataBuffer,1,prefix.ChunkSizeInBytes,BackupFile);
6761|                 if ( NumBytesRead !=
        | prefix.ChunkSizeInBytes ) {
6762|                     Err = PSM_CORRUPT_BACKUP;
6763|                     WRITE_ERROR(L"Error: File '%s'
        | is too short - file is corrupt\n",FileName);
6764|                 } else {
6765|                     ValidChecksum =
        | ValidateChecksum ( prefix.ChunkSizeInBytes, DataBuffer,
        | prefix.ChunkChecksum );
6766|                     if ( !ValidChecksum ) {
6767|                         Err = PSM_CORRUPT_BACKUP;
6768|                         WRITE_ERROR(L"Error: Data
        | checksum failure in '%s' - file is corrupt (chunk
        | number = %d)\n",FileName, ChunkNumber);
6769|                     }
6770|                 }
6771|             }
6772|
6773|             if ( Err == 0 ) {
6774|                 if ( ChunkNumber==1 ) {
6775|                     if ( ContinuationNumber==1 ) {
6776|                         // The first chunk in the
        | first file must be of type VICT_START...
6777|                         if ( prefix.ChunkType !=
        | VICT_START ) {
6778|                             WRITE_ERROR(L"Error:
        | Missing START chunk in '%s' - file is
        | corrupt\n",FileName);
6779|                             Err =

```



```

    | PSM_CORRUPT_BACKUP;
6780|         }
6781|     } else {
6782|         // The first chunk in every
    | other file must be VICT_BEGIN_CONTINUATION
6783|         if ( prefix.ChunkType !=
    | VICT_BEGIN_CONTINUATION ) {
6784|             WRITE_ERROR(L"Error:
    | Missing BEGIN_CONTINUATION chunk in '%s' - file is
    | corrupt\n",FileName);
6785|             Err =
    | PSM_CORRUPT_BACKUP;
6786|         }
6787|     }
6788| }
6789|
6790|     if ( Err == 0 ) {
6791|         FoundFinishChunk = FALSE;
6792|         FoundEndContinuationChunk =
    | FALSE;
6793|         if ( prefix.ChunkType ==
    | VICT_FINISH ) {
6794|             FoundFinishChunk = TRUE;
6795|         } else if ( prefix.ChunkType ==
    | VICT_END_CONTINUATION ) {
6796|             FoundEndContinuationChunk =
    | TRUE;
6797|         }
6798|
6799|         if ( !(prefix.ChunkType &
    | VICT_RESERVED_BIT) ) {
6800|             if (
    | prefix.CumulativeChecksum != *CumulativeChecksum ) {
6801|                 WRITE_ERROR(L"Error:
    | Cumulative checksum failure in '%s' - file is
    | corrupt.\n",FileName);
6802|                 Err =
    | PSM_CORRUPT_BACKUP;
6803|             } else {
6804|                 if ( prefix.ChunkNumber
    | != *ChunkCounter ) {
6805|                     WRITE_ERROR(L"Error: Expected chunk counter %d but
    | found %d in '%s' - file is
    | corrupt.\n",*ChunkCounter,prefix.ChunkNumber,FileName);
6806|                     Err =
    | PSM_CORRUPT_BACKUP;
6807|                 } else {
6808|                     *CumulativeChecksum
    | ^= prefix.PrefixChecksum;

```

```

6809|                                     ++(*ChunkCounter);
6810|                                     }
6811|                                 }
6812|                            }
6813|                        }
6814|                    }
6815|                }
6816|            }
6817|
6818|        free (DataBuffer);
6819|        DataBuffer = NULL;
6820|
6821|        if ( Err == 0 ) {
6822|            if ( ContinuationNumber ==
6823|                | TotalNumberOfFiles ) {
6824|                if ( !FoundFinishChunk ) {
6825|                    WRITE_ERROR(L"Error: Missing
6826|                        | FINISH chunk at end of '%s' - file is
6827|                        | corrupt\n",FileName);
6828|                    Err = PSM_CORRUPT_BACKUP;
6829|                }
6830|            } else {
6831|                if ( !FoundEndContinuationChunk ) {
6832|                    WRITE_ERROR(L"Error: Missing
6833|                        | END_CONTINUATION chunk at end of '%s' - file is
6834|                        | corrupt\n",FileName);
6835|                    Err = PSM_CORRUPT_BACKUP;
6836|                }
6837|            }
6838|        } else {
6839|            WRITE_ERROR(L"Error: Out of memory in file
6840|                | verifier!\n");
6841|            Err = ERROR_OUTOFMEMORY;
6842|        }
6843|        return Err;
6844|    }
6845|
6846|    //-----
6847|    | -----
6848|
6849|    ULONG TestImageFile (
6850|        const WCHAR    *BackupPath,
6851|        int             ContinuationNumber,
6852|        int             TotalNumberOfFiles,
6853|        BOOLEAN         QuickTestOnly,
6854|        ULONG           *CumulativeChecksum,
6855|        ULONG           *ChunkCounter )
6856|    {

```

```

6852|  ULONG Err = PSM_CORRUPT_BACKUP;
6853|  WCHAR FileName [256];
6854|  FILE *BackupFile = NULL;
6855|  ULONG FileError = 0;
6856|
6857|  wcsncpy ( FileName, BackupPath );
6858|  AppendBackslashIfNeeded ( FileName );
6859|  swprintf ( &FileName[wcslen(FileName)],
    | L"image.%03d", ContinuationNumber );
6860|  BackupFile = _w fopen(FileName,L"rb");
6861|  if ( BackupFile ) {
6862|      if ( !QuickTestOnly ) {
6863|          FileError = ChunkFileIsCorrupt (
6864|              FileName,
6865|              BackupFile,
6866|              ContinuationNumber,
6867|              TotalNumberOfFiles,
6868|              CumulativeChecksum,
6869|              ChunkCounter );
6870|      }
6871|      fclose (BackupFile);
6872|      BackupFile = NULL;
6873|      Err = FileError;
6874|  } else {
6875|      WRITE_ERROR(L"Error: Could not open backup
    | file \"%s\\n\", FileName);
6876|  }
6877|
6878|  return Err;
6879| }
6880|
6881| //-----
    | -----
6882|
6883| ULONG TestImageBackup (
6884|     const WCHAR    *BackupPath,
6885|     BOOLEAN        QuickTestOnly )
6886| {
6887|     ULONG Err = PSM_CORRUPT_BACKUP;
6888|     WCHAR FileName [256];
6889|     FILE *SummaryFile = NULL;
6890|     char Line [128];
6891|     char Name [128];
6892|     int Value = 0;
6893|     int NumBackupFiles = 0;
6894|     int ContinuationNumber = 0;
6895|     BOOLEAN FoundCorruptFile = FALSE;
6896|     ULONG CumulativeChecksum = 0;
6897|     ULONG ChunkCounter = 1;
6898|

```

```

6899|  wcscpy ( FileName, BackupPath );
6900|  AppendBackslashIfNeeded ( FileName );
6901|  wscat ( FileName, L"image.000" );
6902|  SummaryFile = _w fopen(FileName,L"rt");
6903|  if ( SummaryFile ) {
6904|      while ( fgets(Line,sizeof(Line),SummaryFile) )
6905|      | {
6906|          if (
6907|          | sscanf(Line,"%[a-zA-Z0-9]=%d",Name,&Value) == 2 ) {
6908|              DEBUG_WRITE(L">>> TestImageBackup:
6909|              | Name='%S', Value=%d\n",Name,Value);
6910|              if ( strcmp(Name,"NumFilesInBackup")==0
6911|              | ) {
6912|                  NumBackupFiles = Value;
6913|              } else if (
6914|              | strcmp(Name,"WarningCode")==0 ) {
6915|                  if ( Value != 0 ) {
6916|                      WRITE(L"Warning Code %08x found
6917|                      | in backup '%s'\n",Value,BackupPath);
6918|                  }
6919|              }
6920|          }
6921|          fclose (SummaryFile);
6922|          SummaryFile = NULL;
6923|          if ( NumBackupFiles > 0 ) {
6924|              ULONG FileError = 0;
6925|              for ( ContinuationNumber=1;
6926|              | ContinuationNumber <= NumBackupFiles;
6927|              | ++ContinuationNumber ) {
6928|                  FileError = TestImageFile (
6929|                  BackupPath,
6930|                  ContinuationNumber,
6931|                  NumBackupFiles,
6932|                  QuickTestOnly,
6933|                  &CumulativeChecksum,
6934|                  &ChunkCounter );
6935|                  if ( FileError != 0 ) {
6936|                      FoundCorruptFile = TRUE;
6937|                      break;
6938|                  }
6939|              }
6940|          if ( !FoundCorruptFile ) {
6941|              if ( QuickTestOnly ) {
6942|                  WRITE(L"All backup image files are
6943|                  | present in '%s'.\n",BackupPath);

```

```

6940|             WRITE(L"Note: The files were not
| tested for internal correctness.\n");
6941|         } else {
6942|             WRITE(L"Backup image is valid in
| '%s'\n",BackupPath);
6943|         }
6944|         Err = 0;
6945|     }
6946| } else {
6947|     WRITE_ERROR(L"Error: Could not find valid
| number of backup files in file '%s'\n",FileName);
6948| }
6949| } else {
6950|     WRITE_ERROR(L"Error: Cannot open file '%s' -
| backup is corrupt\n",FileName);
6951| }
6952|
6953| return Err;
6954| }
6955|
6956| //-----
| -----
6957|
6958| ULONG TestBackupPaths (void)
6959| {
6960|     ULONG Err = 0;
6961|     ULONG NumberOfLocations = 0;
6962|     ULONG OutputImageArrayBytes = MAX_LOCATIONS *
| sizeof(tImageCreationInfo);
6963|     pImageCreationInfo OutputImageArray =
| (pImageCreationInfo) malloc (OutputImageArrayBytes);
6964|
6965|     DEBUG_WRITE(L">>> Entering TestBackupPaths()\n");
6966|
6967|     if ( OutputImageArray ) {
6968|         tLocalLogonInfo LogonInfo = {0};
6969|         LocalLogonCheck (&LogonInfo);
6970|
6971|         Err = PrepareForVolumeImage (
6972|             OutputImageArray,
6973|             MAX_LOCATIONS,
6974|             TRUE,
6975|             &NumberOfLocations );
6976|
6977|         if ( Err == 0 ) {
6978|             WRITE(L"The backup settings in the registry
| are valid.\n");
6979|         } else {
6980|             WRITE_ERROR(L"Error: Backup settings in
| the registry are not valid.\n");

```

```

6981|     }
6982|
6983|     free (OutputImageArray);
6984|     OutputImageArray = NULL;
6985|
6986|     LocalLogoff (&LogonInfo);
6987| } else {
6988|     WRITE_ERROR(L"Error: Out of memory in
        | TestBackupPaths!\n");
6989|     Err = ERROR_OUTOFMEMORY;
6990| }
6991|
6992| DEBUG_WRITE(L">>> TestBackupPaths() returning
        | %08x\n",Err);
6993| return Err;
6994| }
6995|
6996| //-----
        | -----
6997|
6998| ULONG CopyRecoveryFile ( const WCHAR *fSrc, const WCHAR
        | *fDst )
6999| {
7000|     ULONG Err = 0;
7001|     BOOL ItWorked = FALSE;
7002|
7003|     DEBUG_WRITE(L">>> CopyRecoveryFile: source='%s',
        | dest='%s'\n",fSrc,fDst);
7004|
7005|     ItWorked = CopyFileW ( fSrc, fDst, FALSE );
7006|     if ( !ItWorked ) {
7007|         ULONG CopyErr = GetLastError();
7008|         WRITE_ERROR(L">>> Error %08x copying file '%s'
        | to '%s'\n",CopyErr,fSrc,fDst);
7009|         Err = PSM_ERROR_INVALID_PATH;
7010|     }
7011|
7012|     return Err;
7013| }
7014|
7015| //-----
        | -----
7016|
7017| ULONG CopyAllFilesAndDirectory ( const WCHAR *SrcPath,
        | WCHAR *DstPath )
7018| {
7019|     ULONG Err = 0;
7020|     WCHAR FileSpec [256];
7021|     WCHAR wTmpStr [256] = {0};
7022|     WCHAR wTmpStr2[256]={0};

```

```

7023| struct _wfinddata64_t FindData = {0};
7024| long hFile = 0;
7025| int result = 0;
7026|
7027| DEBUG_WRITE(L">>> CopyAllFilesAndDirectory: Source
    | Path='%s'\n",SrcPath);
7028| DEBUG_WRITE(L">>> CopyAllFilesAndDirectory:
    | Destination Path='%s'\n",DstPath);
7029|
7030| wcscpy ( FileSpec, SrcPath );
7031| AppendBackslashIfNeeded (FileSpec);
7032| wcscat ( FileSpec, L"*.*" );
7033| hFile = _wfindfirst64 ( FileSpec, &FindData );
7034| if ( hFile != -1 ) {
7035|     do {
7036|         DEBUG_WRITE(L">>> FindFirst/FindNext:
            | '%s'\n",FindData.name);
7037|         if ( wcscmp(FindData.name,L".")!=0 &&
            | wcscmp(FindData.name,L"..")!=0 ) {
7038|             wcscpy ( FileSpec, SrcPath );
7039|             AppendBackslashIfNeeded (FileSpec);
7040|             wcscat ( FileSpec, FindData.name );
7041|             if ( FindData.attrib & _A_SUBDIR ) {
7042|                 wcscpy(wTmpStr2,DstPath);
7043|                 AppendBackslashIfNeeded (wTmpStr2);
7044|                 wcscat(wTmpStr2,FindData.name);
7045|                 result = _wmkdir(wTmpStr2);
7046|                 if ( result != 0 ) {
7047|                     DEBUG_WRITE(L">>>
                        | _wmkdir('%s')=%d, errno=%d\n",wTmpStr,result,errno);
7048|                 }
7049|                 DEBUG_WRITE(L">>> Doing recursive
                        | call for nested subdirectory...\n");
7050|                 Err = CopyAllFilesAndDirectory
                        | (FileSpec, wTmpStr2);
7051|                 if ( Err != 0 ) {
7052|                     break;
7053|                 }
7054|             } else {
7055|                 wcscpy(wTmpStr,DstPath);
7056|                 AppendBackslashIfNeeded(wTmpStr);
7057|                 wcscat(wTmpStr,FindData.name);
7058|
                        | Err=CopyRecoveryFile(FileSpec,wTmpStr);
7059|                 if ( Err != 0 ) {
7060|                     break;
7061|                 }
7062|             }
7063|         }
7064|     } while ( _wfindnext64(hFile,&FindData)==0 &&

```

```

    | Err==0 );
7065|     _findclose (hFile);
7066| } else {
7067|     DEBUG_WRITE(L">>> FindFirst failed on
    | '%s\n",FileSpec);
7068| }
7069|
7070|
7071|     DEBUG_WRITE(L">>> CopyAllFilesAndDirectory
    | returning %08x\n", Err);
7072|     return Err;
7073| }
7074|
7075| //-----
    | -----
7076|
7077| ULONG GetTimeStamp( WCHAR *ts) {
7078|     ULONG Err=0;
7079|     SYSTEMTIME tNow;
7080|
7081|     GetLocalTime(&tNow);
7082|
7083|
    | swprintf(ts,L"%04u,%02u,%02u,%02u,%02u,%02u",tNow.wYear,
    | tNow.wMonth,tNow.wDay,tNow.wHour,tNow.wMinute,tNow.wSeco
    | nd);
7084|     DEBUG_WRITE(L">>> Creating GetTimeStamp:
    | '%s\n",ts);
7085|     return Err;
7086| }
7087|
7088| //-----
    | -----
7089|
7090| ULONG TestDiskettePath( const WCHAR *dp)
7091| {
7092|     ULONG Err=0;
7093|     WCHAR TempFileName [256];
7094|     FILE *TempFile = 0;
7095|     int result = 0;
7096|
7097|     // Create a temporary file, then erase it, just to
    | see if we can write
7098|     // to the diskette path given.
7099|     wcscpy ( TempFileName, dp );
7100|     AppendBackslashIfNeeded (TempFileName);
7101|     wcscat ( TempFileName, L"vimage.tmp" );
7102|     TempFile = _wfopen(TempFileName,L"wt");
7103|     if ( TempFile ) {
7104|         DEBUG_WRITE(L">>> Successfully opened test file

```



```

    | '%s' for write\n", TempFileName);
7105|     fclose(TempFile);
7106|     TempFile = NULL;
7107|     result = _wremove (TempFileName);
7108|     if ( result == 0 ) {
7109|         DEBUG_WRITE(L">>> Successfully deleted test
    | file '%s'\n", TempFileName);
7110|     } else {
7111|         WRITE_ERROR(L"Error: Could not delete test
    | file '%s'\n", TempFileName);
7112|         Err = PSM_ERROR_INVALID_PATH;
7113|     }
7114| } else {
7115|     WRITE_ERROR(L"Error: Could not write temporary
    | file to path '%s'\n", dp);
7116|     Err = PSM_ERROR_INVALID_PATH;
7117| }
7118| return Err;
7119| }
7120|
7121| //-----
    | -----
7122|
7123| #define ENABLE_bprintf    ULONG PastLabel=FALSE
7124|
7125| #define bprintf(args) \
7126|     if((fprintf args) < 0) {
    | \
7127|         Err=PSM_ERROR_UNSUCCESSFUL;
    | \
7128|         WRITE_ERROR(L"Error writing to
    | '%s'\n",BatchFilePath);    \
7129|         if(!PastLabel) goto PrintError;
    | \
7130|     }
7131|
7132| #define DISABLE_bprintf    PrintError: PastLabel=TRUE
7133|
7134| //-----
    | -----
7135|
7136| WriteBatchFile_AppSets (
7137|     FILE                *BatchFile,
7138|     const WCHAR * const    BatchFilePath,
7139|     const tImageCreationInfo * const    LocationArray,
7140|     ULONG                NumLocations )
7141| {
7142|     ENABLE_bprintf;
7143|     ULONG Err = 0;    // Set by bprintf macro if a
    | write error occurs

```

```

7144|  ULONG ParseErr = 0;
7145|  SYSTEMTIME now = {0};
7146|  ULONG LocationNumber = 0;
7147|  WCHAR Server [256];
7148|  WCHAR ShareOnly [256];
7149|  WCHAR RestOfPath [256];
7150|  WCHAR Drive [16];
7151|  WCHAR NextRemoteDriveLetter = 'Z';
7152|  int LogonIndex = -1;
7153|  BOOLEAN DidLogonCode = FALSE;
7154|
7155|  GetLocalTime(&now);
7156|
7157|  bprintf ((BatchFile, "@echo off\n"));
7158|  bprintf ((BatchFile, "rem call app_logo
| APP_SETS.BAT ***\n"));
7159|  bprintf ((BatchFile, "rem Set NIC over-ride (blank
| for auto-detect)\n"));
7160|  bprintf ((BatchFile, "rem SET NETCARD=\n"));
7161|  bprintf ((BatchFile, "rem SET NOLOGON=\n"));
7162|  bprintf ((BatchFile, "rem If over-riding NIC, or
| updating drivers, copy here\n"));
7163|  bprintf ((BatchFile, "rem copy
| %%RAMD%%\sys%%NETCARD%%.ini
| %%RAMD%%\ini\sys%%NETCARD%%.ini\n"));
7164|  bprintf ((BatchFile, "rem copy
| %%RAMD%%\pro%%NETCARD%%.ini
| %%RAMD%%\ini\pro%%NETCARD%%.ini\n"));
7165|  bprintf ((BatchFile, "\nREM *** Set context for
| this application\n"));
7166|
7167|  wcscpy(Drive,L"X:");
7168|  for ( LocationNumber=0; LocationNumber <
| NumLocations; ++LocationNumber ) {
7169|      bprintf ((BatchFile, "\nREM *** Settings for
| backup location %d\n",1+LocationNumber));
7170|      ParseErr = ParseServerAndShareFromPath (
7171|
| LocationArray[LocationNumber].LogonInfo.Path,
7172|      Server,
7173|      ShareOnly,
7174|      RestOfPath );
7175|
7176|      if ( ParseErr == 0 ) {
7177|          Drive[0] = NextRemoteDriveLetter--;
7178|          if (
| !IsAllWhitespace(LocationArray[LocationNumber].LogonInfo
| .UserId) &&
7179|
| !IsAllWhitespace(LocationArray[LocationNumber].LogonInfo

```

```

    | .Password) &&
7180|         !IsAllWhitespace(Server) ) {
7181|         LogonIndex = LocationNumber;    //
    | remember the first valid server/user/password combo
7182|     }
7183| } else {
7184|     // Looks like a local drive...
7185|     Server[0] = '\0';
7186|     ShareOnly[0] = '\0';
7187|     wcscpy ( RestOfPath,
    | LocationArray[LocationNumber].LogonInfo.Path );
7188|     if ( RestOfPath[1] == ':' ) {
7189|         Drive[0] = RestOfPath[0];
7190|     } else {
7191|         Drive[0] = 'D';    //????
7192|     }
7193| }
7194|
7195|     bprintf ((BatchFile, "SET
    | DRIVE%d=%S\n", 1+LocationNumber, Drive));
7196|     bprintf ((BatchFile, "SET
    | SERVER%d=%S\n", 1+LocationNumber, Server));
7197|     bprintf ((BatchFile, "SET
    | SHARE%d=%S\n", 1+LocationNumber, ShareOnly));
7198|     bprintf ((BatchFile, "SET
    | LOCATION%d=%S\n", 1+LocationNumber, LocationArray[Location
    | Number].LogonInfo.Path));
7199| }
7200|
7201| while ( LocationNumber++ < MAX_LOCATIONS ) {
7202|     bprintf ((BatchFile, "\nREM *** Placeholders
    | for backup location %d\n", LocationNumber));
7203|     bprintf ((BatchFile, "REM SET
    | DRIVE%d=\n", LocationNumber));
7204|     bprintf ((BatchFile, "REM SET
    | SERVER%d=\n", LocationNumber));
7205|     bprintf ((BatchFile, "REM SET
    | SHARE%d=\n", LocationNumber));
7206|     bprintf ((BatchFile, "REM SET
    | LOCATION%d=\n", LocationNumber));
7207| }
7208|
7209|     bprintf ((BatchFile, "\nREM *** Set context for
    | logon\n"));
7210|     if ( LogonIndex >= 0 ) {
7211|         // Getting here means we found a valid
    | server/user/password
7212|         // combo in the list of remote connections.
7213|
7214|         ParseErr = ParseServerAndShareFromPath (

```

```

7215|         LocationArray[LogonIndex].LogonInfo.Path,
7216|         Server,
7217|         ShareOnly,
7218|         RestOfPath );
7219|
7220|     if ( ParseErr == 0 ) {
7221|         WCHAR *p;
7222|         const WCHAR *user;
7223|         const WCHAR *domain;
7224|         WCHAR Buffer[40];
7225|         DidLogonCode = TRUE;
7226|         p =
| wcsrchr(LocationArray[LogonIndex].LogonInfo.UserId,L'\\')
| ;
7227|         if(!p) {
7228|             p =
| wcsrchr(LocationArray[LogonIndex].LogonInfo.UserId,L'/');
7229|         }
7230|         if(p) {
7231|             // user name is in the form
| "domain\name"
7232|             user=p+1;
7233|
| wcsncpy(Buffer,LocationArray[LogonIndex].LogonInfo.UserI
| d,p-LocationArray[LogonIndex].LogonInfo.UserId);
7234|
| Buffer[p-LocationArray[LogonIndex].LogonInfo.UserId]=L'\
| 0';
7235|             domain = Buffer;
7236|         } else {
7237|             user =
| LocationArray[LogonIndex].LogonInfo.UserId;
7238|             domain=NULL;
7239|         }
7240|
7241|         if(domain) {
7242|             bprintf ((BatchFile, "SET
| DOMAIN=%S\n",domain));
7243|             bprintf ((BatchFile, "REM SET
| WORKGROUUP=%S\n\n",domain));
7244|         } else {
7245|             bprintf ((BatchFile, "REM SET
| DOMAIN=psm-domain\n"));
7246|             bprintf ((BatchFile, "REM SET
| WORKGROUUP=psm-domain\n\n"));
7247|         }
7248|         bprintf ((BatchFile, "SET
| SERVER=%S\n",Server));
7249|         bprintf ((BatchFile, "SET
| USER=%S\n",user));

```

```

7250|         bprintf ((BatchFile, "SET
| PASSWORD=%S\n",LocationArray[LogonIndex].LogonInfo.Passw
| ord));
7251|     }
7252| }
7253|
7254| if ( !DidLogonCode ) {
7255|     DidLogonCode = TRUE;
7256|     bprintf ((BatchFile, "REM SET
| DOMAIN=psm-domain\n"));
7257|     bprintf ((BatchFile, "REM SET
| WORKGROUUP=psm-domain\n\n"));
7258|     bprintf ((BatchFile, "REM SET
| SERVER=psm-server\n"));
7259|     bprintf ((BatchFile, "REM SET
| USER=psm-user\n"));
7260|     bprintf ((BatchFile, "REM SET
| PASSWORD=psm-password\n"));
7261| }
7262|
7263| bprintf ((BatchFile, "SET
| NETBIOSNAME=DR%01d%02d%03d\n",(now.wMinute%10),now.wSeco
| nd,now.wMilliseconds));
7264|
7265| DISABLE_bprintf;
7266| DEBUG_WRITE(L">>> WriteBatchFile_AppSets returning
| %08x\n",Err);
7267| return Err;
7268| }
7269|
7270| //-----
| -----
7271|
7272| /*
7273| WriteBatchFile_AppCopy (
7274|     FILE *BatchFile,
7275|     const WCHAR * const BatchFilePath,
7276|     const tImageCreationInfo * const LocationArray,
7277|     ULONG NumLocations )
7278| {
7279|     ENABLE_bprintf;
7280|     ULONG Err = 0; // Set by bprintf macro if a
| write error occurs
7281|
7282|     bprintf ((BatchFile, "@echo off\n"));
7283|     bprintf ((BatchFile, "REM *** Ram drive is ready,
| before file copy, DOS and NET still compressed\n"));
7284|     bprintf ((BatchFile, "SET
| LOGFILE=%%RAMD%%\drlog.txt\n"));
7285|     bprintf ((BatchFile, "call APP_LOGO Logging To

```

```

    | %%LOGFILE%%\n"));
7286|
7287|  DISABLE_bprintf;
7288|  DEBUG_WRITE(L">>> WriteBatchFile_AppCopy returning
    | %08x\n",Err);
7289|  return Err;
7290| }
7291| */
7292|
7293| //-----
    | -----
7294|
7295| WriteBatchFile_AppDoes (
7296|  FILE *BatchFile,
7297|  const WCHAR * const BatchFilePath,
7298|  const tImageCreationInfo * const LocationArray,
7299|  ULONG NumLocations )
7300| {
7301|  ENABLE_bprintf;
7302|  ULONG Err = 0; // Set by bprintf macro if a
    | write error occurs
7303|
7304|  bprintf ((BatchFile, "@ECHO OFF\n"));
7305|  bprintf ((BatchFile, "IF NOT \"%%RESULT%%\"==\"OK\"
    | GOTO FAULT\n"));
7306|  bprintf ((BatchFile, "@rem NOTE: Startup SET's are
    | NOT available here.\n"));
7307|  bprintf ((BatchFile, "@echo Running Disaster
    | Recovery Program...\n"));
7308|  bprintf ((BatchFile, "%%RAMD%%\n"));
7309|  bprintf ((BatchFile, "cd \\\n"));
7310|  bprintf ((BatchFile, "dr /auto\n"));
7311|  bprintf ((BatchFile, "goto end\n"));
7312|  bprintf ((BatchFile, ":FAULT\n"));
7313|  bprintf ((BatchFile, "ECHO Can't run
    | (%%result%%)\n"));
7314|  bprintf ((BatchFile, ":end\n"));
7315|
7316|  DISABLE_bprintf;
7317|  DEBUG_WRITE(L">>> WriteBatchFile_AppDoes returning
    | %08x\n",Err);
7318|  return Err;
7319| }
7320|
7321| //-----
    | -----
7322|
7323| WriteBatchFile_AppLogo (
7324|  FILE *BatchFile,
7325|  const WCHAR * const BatchFilePath,

```

```

7326|  const tImageCreationInfo * const   LocationArray,
7327|  ULONG                               NumLocations )
7328| {
7329|  ENABLE_bprintf;
7330|  ULONG  Err = 0;  // Set by bprintf macro if a
      | write error occurs
7331|
7332|  bprintf((BatchFile,"@echo off\n"));
7333|  bprintf((BatchFile,"echo *** %%1 %%2 %%3 %%4 %%5
      | %%6 %%7 %%8 %%9 >>%%LOGFILE%%\n"));
7334|  bprintf((BatchFile,"cls\n"));
7335|  bprintf((BatchFile,"echo
      | -----\n"));
7336|  bprintf((BatchFile,"echo    Disaster
      | Recovery\n"));
7337| // bprintf((BatchFile,"echo    Version
      | XX.YY.ZZ\n"));
7338|  bprintf((BatchFile,"echo Copyright (c) 2001, CDP,
      | Inc.\n"));
7339|  bprintf((BatchFile,"echo    www.cdp.com\n"));
7340|  bprintf((BatchFile,"echo
      | -----\n"));
7341|  bprintf((BatchFile,"echo *** %%1 %%2 %%3 %%4 %%5
      | %%6 %%7 %%8 %%9\n"));
7342|
7343|  DISABLE_bprintf;
7344|  DEBUG_WRITE(L">>> WriteBatchFile_AppLogo returning
      | %08x\n",Err);
7345|  return Err;
7346| }
7347|
7348| //-----
      | -----
7349|
7350| /*
7351| WriteBatchFile_AppOkay (
7352|  FILE                               *BatchFile,
7353|  const WCHAR * const                BatchFilePath,
7354|  const tImageCreationInfo * const   LocationArray,
7355|  ULONG                               NumLocations )
7356| {
7357|  ENABLE_bprintf;
7358|  ULONG  Err = 0;  // Set by bprintf macro if a
      | write error occurs
7359|
7360|  bprintf((BatchFile,"@ECHO OFF\n"));
7361|  bprintf((BatchFile,"echo *** APP_OKAY.BAT ***\n"));
7362|  bprintf((BatchFile,"echo *** APP_OKAY.BAT
      | ***>>%%LOGFILE%%\n"));
7363|

```

```

7364|  DISABLE_bprintf;
7365|  DEBUG_WRITE(L">>> WriteBatchFile_AppOkay returning
    | %08x\n",Err);
7366|  return Err;
7367| }
7368| */
7369|
7370| //-----
    | -----
7371|
7372| ULONG GenAppBatchFile (
7373|     const WCHAR * const      DestPath,
7374|     const WCHAR * const      BatchFileName,
7375|     BATCH_FILE_WRITER_FUNC    WriteBatchFile,
7376|     const tImageCreationInfo * const  LocationArray,
7377|     ULONG                    NumLocations )
7378| {
7379|     ULONG Err = 0;
7380|     FILE *BatchFile = NULL;
7381|     WCHAR BatchFilePath[256];
7382|     int CloseResult = 0;
7383|
7384|     wcsncpy ( BatchFilePath, DestPath );
7385|     AppendBackslashIfNeeded ( BatchFilePath );
7386|     wcscat ( BatchFilePath, BatchFileName );
7387|     DEBUG_WRITE(L">>> GenAppBatchFile:
    | BatchFilePath='%s\n",BatchFilePath);
7388|
7389|     BatchFile = _wfopen(BatchFilePath,L"wt");
7390|     if ( BatchFile ) {
7391|         Err = (*WriteBatchFile) (BatchFile,
    | BatchFilePath, LocationArray, NumLocations);
7392|         CloseResult = fclose(BatchFile);
7393|         if ( Err==0 && CloseResult!=0 ) {
7394|             WRITE_ERROR(L"Error closing batch file
    | '%s\n",BatchFilePath);
7395|             Err = PSM_ERROR_UNSUCCESSFUL;
7396|         }
7397|         BatchFile = NULL;
7398|     } else {
7399|         WRITE_ERROR(L"Error opening batch file '%s' for
    | write\n",BatchFilePath);
7400|     }
7401|
7402|     DEBUG_WRITE(L">>> GenAppBatchFile: returning
    | %08x\n",Err);
7403|     return Err;
7404| }
7405|
7406| //-----

```



```

| -----
7407|
7408| ULONG GenerateRecoveryDisketteAppFiles ( const WCHAR *
    | const _DestPath )
7409| {
7410|     ULONG Err = 0;
7411|     ULONG LocationErr = 0;
7412|     ULONG LocationIndex = 0;
7413|     ULONG NumLocations = 0;
7414|     WCHAR DestPath [256];
7415|     WCHAR BackupName [64];
7416|     WCHAR ImageName [256];
7417|     tImageCreationInfo LocationArray [MAX_LOCATIONS] =
    | {0};
7418|     HKEY Key = INVALID_HANDLE_VALUE;
7419|     ULONG BatchFileIndex = 0;
7420|
7421|     struct sBatchFileEntry {
7422|         BATCH_FILE_WRITER_FUNC WriterFunc;
7423|         const WCHAR * const BatchFileName;
7424|     } BatchFileArray[] = {
7425|         { WriteBatchFile_AppSets, L"app_sets.bat" },
7426| //     { WriteBatchFile_AppCopy, L"app_copy.bat" },
7427|         { WriteBatchFile_AppDoes, L"app_does.bat" },
7428|         { WriteBatchFile_AppLogo, L"app_logo.bat" },
7429| //     { WriteBatchFile_AppOkay, L"app_okay.bat" },
7430|         { NULL, NULL } // marks end of list
7431|     };
7432|
7433|     wcscpy ( DestPath, _DestPath );
7434|     AppendBackslashIfNeeded (DestPath);
7435|     wscat ( DestPath, L"app\\" );
7436|     _wmkdir(DestPath);
7437|     DEBUG_WRITE(L">>> GenerateRecoveryDisketteAppFiles:
    | DestPath='%s'\n",DestPath);
7438|
7439|     Err = RegOpenKeyExW(
7440|         HKEY_LOCAL_MACHINE, // handle of open key
7441|         BackupRegistryPath, // address of name of
    | subkey to open
7442|         0, // reserved
7443|         KEY_READ, // security access mask
7444|         &Key ); // address of handle of
    | open key
7445|
7446|     if ( Err == 0 ) {
7447|         ULONG DataSize = sizeof(ImageName);
7448|         Err = RegQueryValueExW(
7449|             Key, // handle of key to
    | query

```

```

7450|         L"ImageName",          // address of name
      | of value to query
7451|         NULL,                  // reserved
7452|         NULL,                  // address of
      | buffer for value type
7453|         (char*)ImageName,      // address of data
      | buffer
7454|         &DataSize );          // address of data
      | buffer size
7455|
7456|     if ( Err == 0 ) {
7457|         for ( LocationIndex=1; LocationIndex <=
      | MAX_LOCATIONS; ++LocationIndex ) {
7458|             swprintf ( BackupName, L"Backup%d",
      | LocationIndex );
7459|             LocationErr = GetSettingsForBackup (
      | Key, BackupName, &LocationArray[NumLocations] );
7460|             if ( LocationErr == 0 ) {
7461|                 if ( !IsAllWhitespace
      | (LocationArray[NumLocations].LogonInfo.Path) ) {
7462|                     AppendBackslashIfNeeded (
      | LocationArray[NumLocations].LogonInfo.Path );
7463|                     wscat (
      | LocationArray[NumLocations].LogonInfo.Path, ImageName
      | );
7464|                     DEBUG_WRITE(L">>>
      | GenerateRecoveryDisketteAppFiles: path %d =
      | '%s'\n",1+NumLocations,LocationArray[NumLocations].Logon
      | Info.Path);
7465|
      | LocationArray[NumLocations].LocationNumber =
      | LocationIndex;
7466|                 ++NumLocations;
7467|             }
7468|         }
7469|     }
7470|
7471|     if ( NumLocations > 0 ) {
7472|         for ( BatchFileIndex=0; Err==0 &&
      | BatchFileArray[BatchFileIndex].WriterFunc;
      | ++BatchFileIndex ) {
7473|             Err = GenAppBatchFile (
7474|                 DestPath,
7475|
      | BatchFileArray[BatchFileIndex].BatchFileName,
7476|
      | BatchFileArray[BatchFileIndex].WriterFunc,
7477|                 LocationArray,
7478|                 NumLocations );
7479|         }

```

```

7480|
7481|         if ( Err == 0 ) {
7482|             DEBUG_WRITE(L">>>
| GenerateRecoveryDisketteAppFiles: All batch files
| created successfully\n");
7483|         }
7484|     } else {
7485|         WRITE_ERROR(L"Error: Could not find
| any valid backup locations in registry.\n");
7486|         Err = PSM_ERROR_UNSUCCESSFUL;
7487|     }
7488| } else {
7489|     WRITE_ERROR(L"Error %08x getting value of
| 'ImageName' from registry key
| '%s'\n",Err,BackupRegistryPath);
7490| }
7491|
7492|     RegCloseKey(Key);
7493|     Key = INVALID_HANDLE_VALUE;
7494| } else {
7495|     WRITE_ERROR(L"Error %08x getting backup
| registry settings.\n",Err);
7496| }
7497|
7498|     DEBUG_WRITE(L"GenerateRecoveryDisketteAppFiles
| returning %08x\n",Err);
7499|     return Err;
7500| }
7501|
7502| //-----
| -----
7503|
7504| ULONG CreateRecoveryDiskette (void)
7505| {
7506|     ULONG Err = 0;
7507|     ULONG DataSize=256;
7508|     WCHAR wRecoveryPath[256]={0};
7509|     WCHAR wts[256]={0};
7510|     WCHAR wTmpStr[256]={0};
7511|     struct _wfinddatai64_t FindData={0};
7512|     long hFile=0;
7513|     DWORD RecoveryResult=PSM_ERROR_UNSUCCESSFUL;
7514|     HKEY Key = INVALID_HANDLE_VALUE;
7515|     ULONG StringLength = 0;
7516|     const WCHAR * const TailPattern = L"ss.exe";
7517|     const ULONG TailPatternLength =
| wcslen(TailPattern);
7518|     tRemoteLogonInfo RemoteLogonInfo = {0};
7519|     tLocalLogonInfo LocalLogonInfo = {0};
7520|

```

```

7521| // open the registry
7522| Err = RegOpenKeyExW(
7523|     HKEY_LOCAL_MACHINE, // handle of open key
7524|     BackupRegistryPath, // address of name of
    | subkey to open
7525|     0, // reserved
7526|     KEY_READ, // security access mask
7527|     &Key ); // address of handle of
    | open key
7528|
7529| if ( Err == 0 ) {
7530|     LocalLogonCheck (&LocalLogonInfo);
7531|
7532|     // Get The Backup Initiator Path
7533|     DataSize = sizeof(wRecoveryPath);
7534|     Err = RegQueryValueExW(
7535|         Key, // handle of key to
    | query
7536|         L"BackupInitiator", // address of name
    | of value to query
7537|         NULL, // reserved
7538|         NULL, // address of
    | buffer for value type
7539|         (char*)wRecoveryPath, // address of data
    | buffer
7540|         &DataSize ); // address of data
    | buffer size
7541|
7542|     RegCloseKey(Key); // Closes Backup Key
7543|     Key = INVALID_HANDLE_VALUE;
7544|
7545|     if (Err == 0 ) {
7546|         // Setup Path to Recovery Diskette
    | Directory
7547|
7548|         // We figure out the path to the recovery
    | diskette image based on
7549|         // the path to the BackupInitiator. In
    | other words, the RecoveryDiskette
7550|         // subdirectory must be in the same place
    | as ss.exe.
7551|
7552|         wcscpy (wTmpStr, wRecoveryPath);
7553|         StringLength = ExpandEnvironmentStringsW
    | (wTmpStr,wRecoveryPath,sizeof(wRecoveryPath)/sizeof(wRec
    | overyPath[0]));
7554|         DEBUG_WRITE(L">>> ExpandEnvironmentStringsW
    | returned %d,
    | wRecoveryPath='%s'\n",StringLength,wRecoveryPath);
7555|         if ( StringLength>TailPatternLength &&

```

```

| _wcsicmp(&wRecoveryPath[StringLength-TailPatternLength-1
| ],TailPattern)==0 ) {
7556|         --StringLength;    // we don't want
| null terminator included in the length
7557|
| wRecoveryPath[StringLength-TailPatternLength]='\0';
7558|
| wcscat(wRecoveryPath,L"RecoveryDiskette\\");
7559|         DEBUG_WRITE(L">>> Obtained diskette
| path '%s' from registry value
| '%s\\n",wRecoveryPath,wTmpStr);
7560|
7561|         // Open Diskette SubKey
7562|         wcscpy(wTmpStr,BackupRegistryPath);
7563|         wcscat(wTmpStr,L"Diskette");
7564|         Err = RegOpenKeyExW(
7565|             HKEY_LOCAL_MACHINE,    // handle
| of open key
7566|             wTmpStr,                // address
| of name of subkey to open
7567|             0,                      // reserved
7568|             KEY_READ,               // security
| access mask
7569|             &Key );                // address
| of handle of open key
7570|
7571|         DataSize =
| sizeof(RemoteLogonInfo.Path);
7572|         if (Err == 0) {
7573|             Err = RegQueryValueExW(
7574|                 Key,                //
| handle of key to query
7575|                 L"DiskettePath",    //
| address of name of value to query
7576|                 NULL,               //
| reserved
7577|                 NULL,               //
| address of buffer for value type
7578|                 (char*)RemoteLogonInfo.Path,
| // address of data buffer
7579|                 &DataSize );        //
| address of data buffer size
7580|
7581|         if (Err==0) {
7582|             RegCloseKey(Key); // Closes
| Diskette SubKey
7583|             Err =
| GetSettingsForRecoveryDisketteCreation (
| &RemoteLogonInfo );
7584|             if ( Err == 0 ) {

```

```

7585|                Err = RemoteLogonCheck
       | (&RemoteLogonInfo);
7586|                if ( Err == 0 ) {
7587|                    Err =
       | TestDiskettePath(RemoteLogonInfo.Path);
7588|                if (Err == 0) {
7589|
       | Err=TestDiskettePath(wRecoveryPath);
7590|                if (Err==0) {
7591|
       | Err=GetTimeStamp(wts);
7592|                UpdateTimeStamp
       | ( wTmpStr, L"LastUpdate", wts );
7593|
       | UpdateBackupStatus( wTmpStr, L"LastUpdateResult",
       | PSM_RUNNING);
7594|                // Copy
       | Recovery Files
7595|                Err =
       | CopyAllFilesAndDirectory ( wRecoveryPath,
       | RemoteLogonInfo.Path );
7596|                if (Err ==0) {
7597|                    Err =
       | GenerateRecoveryDisketteAppFiles ( RemoteLogonInfo.Path
       | );
7598|                if ( Err ==
       | 0 ) {
7599|
       | RecoveryResult=PSM_OPERATION_SUCCESSFUL;
7600|                }
7601|                }
7602|                // Write RESULT
       | to Diskette SubKey
7603|
       | UpdateBackupStatus ( wTmpStr, L"PathStatus",
       | PSM_VALID_PATH );
7604|
       | UpdateBackupStatus ( wTmpStr, L"LastUpdateResult",
       | RecoveryResult );
7605|                } // End If Not a
       | Valid Recovery Path
7606|                else {
7607|
       | UpdateBackupStatus ( wTmpStr, L"PathStatus",
       | PSM_VALID_PATH);
7608|
       | UpdateBackupStatus ( wTmpStr, L"LastUpdateResult",
       | PSM_ERROR_INVALID_PATH );
7609|                }
7610|

```

```

7611|             RemoteLogoff (
    | &RemoteLogonInfo );
7612|         }
7613|     else {
7614|         UpdateBackupStatus
    | ( wTmpStr, L"PathStatus", Err );
7615|         UpdateBackupStatus
    | ( wTmpStr, L"LastUpdateResult", RecoveryResult );
7616|     }
7617| } else {
7618|     WRITE_ERROR(L"FAIL:
    | Error %08x connecting to diskette destination
    | '%s'\n",Err,RemoteLogonInfo.Path);
7619|     UpdateBackupStatus (
    | wTmpStr, L"PathStatus", Err );
7620|     UpdateBackupStatus (
    | wTmpStr, L"LastUpdateResult", RecoveryResult );
7621| }
7622| } else {
7623|     WRITE_ERROR(L"FAIL: Error
    | %08x getting registry settings for remote
    | logon\n",Err);
7624| }
7625| } // End if Registry Could Read
    | Diskette Path
7626| else {
7627|     RegCloseKey(Key); // Closes
    | Diskette SubKey
7628|     WRITE_ERROR(L"FAIL: Couldn't
    | Access Diskette Path.\n");
7629| }
7630| } // End if Valid Diskette SubKey
7631| else {
7632|     WRITE_ERROR(L"FAIL: Couldn't Access
    | Diskette Registry SubKey.\n");
7633| }
7634| } // End if StringLength >
    | TailPatternLength
7635| else {
7636|     WRITE_ERROR(L"FAIL: Invalid
    | BackupInitiator registry value '%s'\n", wTmpStr);
7637|     Err = PSM_ERROR_INVALID_PARAMETER;
7638| }
7639| } // End if Valid Backup Initiator Key Read
7640| else {
7641|     WRITE_ERROR(L"FAIL: Could not access
    | BackupInitiator value in key '%s'\n",
    | BackupRegistryPath);
7642| }
7643|

```

```

7644|     LocalLogoff (&LocalLogonInfo);
7645| } // End if Could Open Backup Registry Key
7646| else {
7647|     WRITE_ERROR(L"FAIL: Couldn't Access Backup
    | Registry Key.\n");
7648| }
7649| if (RecoveryResult==PSM_OPERATION_SUCCESSFUL) {
7650|     WRITE(L"Recovery diskette creation
    | Completed.\n");
7651| }
7652| else {
7653|     WRITE_ERROR(L"Recovery diskette creation
    | Failed.\n");
7654| }
7655|
7656| return Err;
7657| }
7658|
7659| /*--- end of file vimage.c ---*/
7660|
7661|
7662|
7663| File Listing: File: vimage.h
7664|
7665| #define DR_BACKUP_SUPPORTED    1
7666|
7667| //-----
    | -----
7668| // Volume Image Chunk Types...
7669| #define VICT_UNDEFINED        0
7670| #define VICT_START            1
7671| #define VICT_GRANULE          2
7672| #define VICT_FINISH           3
7673| #define VICT_PARTITION_INFO    4
7674|
7675| // Volume Image Internal Chunk Types
7676| // If VICT_RESERVED_BIT is set in a chunk type, it
    | means that the chunk type
7677| // is for internal chunk file management and should not
    | be used by anything else.
7678| #define VICT_RESERVED_BIT      0x8000
7679| #define VICT_BEGIN_CONTINUATION (VICT_RESERVED_BIT
    | | 1)
7680| #define VICT_END_CONTINUATION  (VICT_RESERVED_BIT
    | | 2)
7681|
7682| //-----
    | -----
7683| #define VI_COMPRESS_UNDEFINED    0
7684| #define VI_COMPRESS_NONE        1

```



```

7685| #define VI_COMPRESS_ALL_BYTES_SAME 2
7686|
7687| //-----
7688| | -----
7689| #define VOLIMAGE_MAX_USER_DATA 20
7690|
7691| typedef struct sVolumImage_ChunkPrefix {
7692| | //-----
7693| | -----
7694| | // The following checksums *MUST* remain at the
7695| | beginning of the structure!
7696| | ULONG PrefixChecksum; // checksum of
7697| | prefix, starting after ChunkChecksum
7698| | ULONG ChunkChecksum; // checksum of data
7699| | after the prefix
7700| | //-----
7701| | -----
7702| | ULONG ChunkType;
7703| | ULONG PrefixSizeInBytes; //
7704| | sizeof(tVolumImage_ChunkPrefix)
7705| | ULONG ChunkSizeInBytes; // size
7706| | of data following this prefix
7707| | ULONG UserData [VOLIMAGE_MAX_USER_DATA]; // may
7708| | be used on a per chunk type basis
7709| | ULONG CumulativeChecksum; // the
7710| | XOR of all non-continuation prefix checksums before
7711| | this chunk in backup set
7712| | ULONG ChunkNumber; //
7713| | ordinal number of this chunk in the backup set,
7714| | starting at 1
7715| | ULONG Reserved [5]; //
7716| | reserved for internal chunk use
7717| } tVolumImage_ChunkPrefix, *pVolumImage_ChunkPrefix;
7718|
7719| typedef struct sVolumImage_Chunk {
7720| | tVolumImage_ChunkPrefix Prefix;
7721| | char Data[0];
7722| } tVolumImage_Chunk, *pVolumImage_Chunk;
7723|
7724| //-----
7725| | -----
7726|
7727| ULONG UpdateEngineStatus ( ULONG Status ); // updates
7728| | psman5\Backup\EngineStatus reg value
7729|
7730| ULONG DoVolumImageBackup (

```

```

7717|  const WCHAR    *VolumeName,
7718|  const WCHAR    *OriginalVolumeName,
7719|  PVOID          AbortEvent );
7720|
7721| //-----
7722| | -----
7723| ULONG TestImageBackup (
7724|  const WCHAR    *BackupPath,
7725|  BOOLEAN        QuickTestOnly );
7726|
7727| ULONG CreateRecoveryDiskette (void);
7728| ULONG TestBackupPaths (void);
7729|
7730| //-----
7731| | -----
7732| #define CHECKSUM_IGNORE_DWORD 0xf5e4d3c1
7733|
7734|
7735| ULONG CalculateChecksum (
7736|  ULONG          DataSizeInBytes,
7737|  const void     *DataBuffer );
7738|
7739| BOOLEAN ValidateChecksum (
7740|  ULONG          DataSizeInBytes,
7741|  const void     *DataBuffer,
7742|  ULONG          StoredChecksum );
7743|
7744| /*--- end of file vimage.h ---*/
7745|
7746|
7747|
7748| File Listing: File: vimage.inc
7749|
7750| /*
7751|  * Source code included by ss.c to either define or
7752|  | stub out DoSnapShotBackup function.
7753|  *
7754|  */
7755| int DoSnapShotBackup ( const WCHAR *VolumeName )
7756| {
7757|  int Err = 0;
7758|  SECURITY_ATTRIBUTES sa={0};
7759|  PSECURITY_DESCRIPTOR sd={0};
7760|  HANDLE EventHandle = INVALID_HANDLE_VALUE;
7761|  ULONG EventResult = 0;
7762|
7763|  sd =

```

```

    | GetSecuritySD(FALSE,EVENT_ALL_ACCESS,EVENT_ALL_ACCESS,0)
    | ;
7764|     if(!sd) {
7765|         Err = GetLastError();
7766|         WRITE_ERROR(L"Error %08x getting
    | security\n",Err);
7767|         return Err;
7768|     }
7769|
7770|     sa.nLength = sizeof(SEURITY_ATTRIBUTES);
7771|     sa.bInheritHandle = FALSE;
7772|     sa.lpSecurityDescriptor = sd;
7773|
7774|     EventHandle = CreateEventW ( &sa, TRUE, FALSE,
7775|
    | L"Global\PersistentStorageManager_DisasterRecovery_Back
    | up" );
7776|
7777|     EventResult = GetLastError();
7778|
7779|     if ( EventHandle == NULL ) {
7780|         if ( EventResult == ERROR_ACCESS_DENIED ) {
7781|             WRITE_ERROR(L"Error: A backup is already
    | in progress on this machine\n");
7782|             Err = PSM_BACKUP_ALREADY_IN_PROGRESS;
7783|         } else {
7784|             WRITE_ERROR(L"Error: Could not create
    | backup event (%08x)\n",EventResult);
7785|         }
7786|         Err = EventResult;
7787|     } else {
7788|         if ( EventResult == ERROR_ALREADY_EXISTS ) {
7789|             Err = PSM_BACKUP_ALREADY_IN_PROGRESS;
7790|             WRITE_ERROR(L"Error: A backup is already
    | in progress on this machine\n");
7791|             DEBUG_WRITE(L">>> (Backup event was already
    | set.)\n" );
7792|         } else {
7793|             if ( VolumeName[0] && !VolumeName[1] ) {
7794|                 // If it's a single-letter name, assume
    | it's a drive to take a snapshot of...
7795|                 swprintf ( InVolumeMapData[0],
    | L"%c:\\", toupper(VolumeName[0]) );
7796|                 if (
    | !Psm_CanBePSMedW(InVolumeMapData[0]) ) {
7797|                     Err = 1;
7798|                     WRITE_ERROR(L"Volume '%s' cannot be
    | PSMed\n",InVolumeMapData[0]);
7799|                 } else {
7800|

```

```

7801|             InVolumeMap[0] =
| InVolumeMapData[0];
7802|             VolumeMapFlags[0] = PSM_VOLUME_MAP;
7803|             NumVolumes = 1;
7804|
7805|             | memset(&In,0,sizeof(tOpenTransactionInPersistentW));
7806|
7807|             In.Size =
| sizeof(tOpenTransactionInPersistentW);
7808|             Err =
| GetSnapShotCreationFlags(SnapShotFlags, &In.Flags);
7809|             if ( Err == 0 ) {
7810|                 In.CallerPrivateUse =
| (PVOID)GroupNumber;
7811|                 In.NumToKeep      =
| NumToKeep;
7812|
7813|                 // this is the snapshot name or
| L"" if none
7814|                 wcsncpy(In.SnapShotName,L"System
| Backup");
7815|
7816|                 // not used
7817|                 In.ErrorEvent      = NULL;
7818|
7819|                 // set event so they can cancel
| waiting for q period
7820|                 // pass in a null dacl so
| anyone can access it, otherwise
7821|                 // passing in null, gives it
| "default" which doesnt allow normal users
7822|                 // also, create in the global
| namespace instead of the session namespace
7823|                 AbortEvent = In.AbortEvent =
| CreateEvent( &sa, TRUE, FALSE,
| TEXT("Global\\PSM_Backup_Abort_Event"));
7824|
7825|                 // Set our Control - C handler
7826|                 wcsncpy(AbortMessage,L",
| cancelling creation of snapshot.\n");
7827|                 SetConsoleCtrlHandler(
| (PHANDLER_ROUTINE)CtrlHandlerRoutine, TRUE );
7828|
7829|                 In.Priority = (unsigned
| char)PRIORITY_HIGHEST;
7830|                 In.SnapShotFlags =
| SnapShotFlags;
7831|
7832|                 OutVolumeMapByteSize = 32768;

```

```

7833|             OutVolumeMap =
| malloc(OutVolumeMapByteSize);
7834|             if(OutVolumeMap) {
7835|                 WRITE(L"Creating snapshot
| for backup...\n");
7836|                 UpdateEngineStatus
| (PSM_CREATING_SNAPSHOT);
7837|                 SnapShot = NULL;
7838|                 Err = Psm_CreateSnapShotW(
7839|                     (pOpenTransactionInW)&In,
7840|                     NumVolumes,
7841|                     InVolumeMap,
7842|                     VolumeMapFlags,
7843|                     OutVolumeMapByteSize,
7844|                     OutVolumeMap,
7845|                     &Out,
7846|                     &SnapShot,
7847|                     NULL,
7848|                     NULL );
7849|
7850|                 if(!Err) {
7851|                     if ( SnapShot ) {
7852|                         WRITE(L"Snapshot
| %08x created successfully\n", (ULONG)SnapShot);
7853|                         WRITE(L"'%s' =
| '%s\n", InVolumeMap[0], OutVolumeMap[0]);
7854|                         | wcsncpy(AbortMessage, L", cancelling backup.\n");
7855|                         Err =
| DoVolumeImageBackup
| (OutVolumeMap[0], InVolumeMapData[0], AbortEvent);
7856|                         UpdateEngineStatus
| ( PSM_DESTROYING_SNAPSHOT );
7857|                         | DeleteExistingSnapShot ( (ULONG)SnapShot );
7858|                         } else {
7859|                         | WRITE_ERROR(L"Internal error: SnapShot==NULL after
| Psm_CreateSnapShotW returned 0\n");
7860|                         }
7861|                         } else {
7862|                         WRITE_ERROR(L"Error
| %08x creating snapshot\n", Err);
7863|                         }
7864|
7865|                         free(OutVolumeMap);
7866|                         OutVolumeMap = NULL;
7867|                     } else {
7868|                         WRITE_ERROR(L"Out of

```

```

    | memory\n");
7869|         Err = ERROR_OUTOFMEMORY;
7870|     }
7871|
7872|         if(AbortEvent) {
7873|             CloseHandle(AbortEvent);
7874|             AbortEvent =
    | INVALID_HANDLE_VALUE;
7875|         }
7876|     } else {
7877|         WRITE_ERROR(L"Error: Invalid
    | snapshot creation flags.\n");
7878|     }
7879|
7880|         if(Err) {
7881|             PrintWin32Error(Err);
7882|         }
7883|     }
7884| } else {
7885|     // Assume they directly want to back up
    | a volume, not via a snapshot...
7886|     // set event so they can cancel waiting
    | for q period
7887|     // pass in a null dacl so anyone can
    | access it, otherwise
7888|     // passing in null, gives it "default"
    | which doesnt allow normal users
7889|     // also, create in the global namespace
    | instead of the session namespace
7890|     AbortEvent = CreateEvent( &sa, TRUE,
    | FALSE, TEXT("Global\\PSM_Backup_Abort_Event"));
7891|
7892|     // Set our Control - C handler
7893|     wcsncpy(AbortMessage,L", cancelling
    | backup.\n");
7894|     SetConsoleCtrlHandler(
    | (PHANDLER_ROUTINE)CtrlHandlerRoutine, TRUE );
7895|     Err = DoVolumeImageBackup (VolumeName,
    | VolumeName, AbortEvent);
7896|     if(AbortEvent) {
7897|         CloseHandle(AbortEvent);
7898|         AbortEvent = INVALID_HANDLE_VALUE;
7899|     }
7900|     if(Err) {
7901|         PrintWin32Error(Err);
7902|     }
7903| }
7904| }
7905| CloseHandle (EventHandle);
7906| EventHandle = NULL;

```

```

7907|    }
7908|
7909|    if(sd) {
7910|        LocalFree(sd);
7911|    }
7912|
7913|    UpdateEngineStatus (PSM_IDLE);
7914|    return Err;
7915| }
7916|
7917|
7918|
7919| PSM Disaster Recovery Backup Source
7920|
7921| Backup creates an image of the current system volume on
    | up to three local hard drives or network share volumes.
    | --LPW
7922|
7923|
7924|
7925| File Listing: drbackup.c
7926|
7927| /*
7928|  * This is the main source file for drbackup.dll,
7929|  * the PSM Disaster Recovery Backup module.
7930|  */
7931|
7932| #include <stdio.h>
7933| #include <stdlib.h>
7934| #include <string.h>
7935| #include <stddef.h>
7936| #include <windows.h>
7937| #include <tchar.h>
7938| #include <process.h>
7939| #include <time.h>
7940| #include <direct.h>
7941| #include <lm.h>
7942| #include <assert.h>
7943| #define ASSERT assert
7944|
7945| #include <winioctl.h>
7946| #include <undoc.h>
7947| // psm api
7948| #include <psm.h>
7949| // ioctls we need to send down
7950| #include "..\..\driver\ioctl.h"
7951|
7952| #include "volume.h"
7953|
7954| #include "defrag.h"

```

```

7955| #include <mountdev.h>
7956| #include <ntddstor.h>
7957| #include <ntddvol.h>
7958| #include <aclapi.h>
7959| #include <clusapi.h>
7960|
7961| #include "setup5.h"
7962| #include "setup4.h"
7963| #include "service.h"
7964| #include "cluster.h"
7965|
7966| #ifdef DEBUG
7967|     #define STATIC
7968| #else
7969|     #define STATIC static
7970| #endif
7971|
7972| #define STATUS_SUCCESS ((ULONG)0)
7973|
7974| #ifdef WIN32
7975| #define DLLEXPORT __declspec( dllexport )
7976| #define DLLIMPORT __declspec( dllimport )
7977| #endif
7978|
7979| //-----
7980| | -----
7981| | -----
7982| typedef struct sThreadStorage {
7983|     HINSTANCE      hInstance;
7984|     DWORD          fdwReason;
7985| } tThreadStorage, *pThreadStorage;
7986|
7987| //-----
7988| | -----
7989| | -----
7990| STATIC DWORD TlsIndex = 0xffffffff;
7991|
7992| //-----
7993| | -----
7994| | -----
7995| STATIC void InitThreadLocalStorage ( pThreadStorage
7996|     | ThreadStorage )
7997| {
7998|     memset (ThreadStorage, 0, sizeof(tThreadStorage));
7999| }
8000|
8001|
8002| //-----

```



```

| -----
| -----
7998|
7999| /*
8000| When a process uses load-time linking with this DLL,
    | the entry-point
8001| function is sufficient to manage the thread local
    | storage. Problems can
8002| occur with a process that uses run-time linking because
    | the entry-point
8003| function is not called for threads that exist before
    | the LoadLibrary
8004| function is called, so TLS memory is not allocated for
    | these threads.
8005| The following example solves this problem by checking
    | the value returned
8006| by the TlsGetValue function and allocating memory if
    | the value indicates
8007| that the TLS slot for this thread is not set.
8008| */
8009| STATIC pthreadStorage_t GetThreadStorage()
8010| {
8011|     pthreadStorage_t ThreadStorage = (pthreadStorage_t)
        | TlsGetValue(TlsIndex);
8012|
8013|     // If NULL, allocate memory for this thread.
8014|
8015|     if (ThreadStorage == NULL) {
8016|         ThreadStorage = LocalAlloc(LPTR,
            | sizeof(pthreadStorage_t));
8017|         if ( ThreadStorage ) {
8018|             TlsSetValue(TlsIndex, ThreadStorage);
8019|             InitThreadLocalStorage (ThreadStorage);
8020|         }
8021|     }
8022|     return ThreadStorage;
8023| }
8024|
8025| //-----
    | -----
    | -----
8026|
8027| STATIC WCHAR NibbleToHexWChar( unsigned char In,
    | BOOLEAN TakeUpper )
8028| {
8029|     In = TakeUpper?(In >> 4):(In & 0xf);
8030|     return (WCHAR)( ( In > 9 )?(In - 10 + 'a):(In +
        | '0') );
8031| }
8032|

```

```

8033|
8034| STATIC ULONG BufferToHexWChar( PVOID Buffer, ULONG
    | NumBytes, PWCHAR Out, ULONG *OutSize)
8035| {
8036|     ULONG Status = 0;
8037|     ULONG i;
8038|     unsigned char *In = (unsigned char*)Buffer;
8039|
8040|     //Round to take whole number of D_words
8041|     NumBytes = (NumBytes+3)/4*4;
8042|
8043|     if (Out==NULL) {
8044|         *OutSize = (NumBytes*2+1)*sizeof(WCHAR);
8045|     } else {
8046|         if (*OutSize < 2) {
8047|             // just give back bad status as not room to
            | put even an empty string
8048|             Status = ERROR_INSUFFICIENT_BUFFER;
8049|
8050|         } else {
8051|
8052|             if ((NumBytes*2+1)*sizeof(WCHAR) > *OutSize
            | ) {
8053|                 //not enough room for all we have pack
            | the limit and give bad status
8054|                 NumBytes =
            | (*OutSize/sizeof(WCHAR)-1)/2;
8055|                 Status = ERROR_BUFFER_OVERFLOW;
8056|             }
8057|             *OutSize = (NumBytes*2+1)*sizeof(WCHAR);
8058|
8059|             for (i=0;i<NumBytes;In++,i++,Out+=2) {
8060|                 Out[0] = NibbleToHexWChar(In[0],1);
8061|                 Out[1] = NibbleToHexWChar(In[0],0);
8062|             }
8063|             Out[0] = L'\0';
8064|         }
8065|     }
8066|     return Status;
8067| }
8068|
8069|
8070|
8071| STATIC ULONG GetVolumeIdentifiers (
8072|     const WCHAR    *OriginalVolumeName,
8073|     HANDLE          VolumeHandle,
8074|     DWORD           *SerialNumber,
8075|     DWORD           *Uniqueld )
8076| {
8077|     ULONG Err = 0;

```

```

8078|  BOOL ReadGood = GetVolumeInformationW (
8079|      OriginalVolumeName,    // root directory
8080|      NULL,                  // volume name buffer
8081|      0,                    // length of name
      | buffer
8082|      SerialNumber,          // volume serial number
8083|      NULL,                  // maximum file name
      | length
8084|      NULL,                  // file system options
8085|      NULL,                  // file system name
      | buffer
8086|      0 );                  // length of file
      | system name buffer
8087|
8088|  if ( !ReadGood ) {
8089|      Err = GetLastError();
8090|  } else {
8091|      HANDLE OriginalVolumeHandle =
      | INVALID_HANDLE_VALUE;
8092|      BYTE UniqueRaw [256] = {0};
8093|      PMOUNTDEV_UNIQUE_ID UniquePtr =
      | (PMOUNTDEV_UNIQUE_ID) UniqueRaw;
8094|      ULONG dwBytesReturned = 0;
8095|      ULONG Length = 0;
8096|      WCHAR OriginalVolumePath [256] = {0};
8097|
8098|      wcscpy ( OriginalVolumePath, L"\\.\\" );
8099|      wcscat ( OriginalVolumePath, OriginalVolumeName
      | );
8100|      Length = wcslen(OriginalVolumePath);
8101|      if ( Length>0 &&
      | OriginalVolumePath[Length-1]=='\' ) {
8102|          OriginalVolumePath[--Length] = 0;
8103|      }
8104|
8105|      OriginalVolumeHandle = CreateFileW(
8106|          OriginalVolumePath,    //
      | file name
8107|          GENERIC_READ,          // access
      | mode
8108|          FILE_SHARE_READ|FILE_SHARE_WRITE,
      | // share mode
8109|          NULL,                  // SD
8110|          OPEN_EXISTING,         // how to
      | create
8111|          FILE_ATTRIBUTE_NORMAL,
      | // file attributes
8112|          NULL );                // handle
      | to template file
8113|

```

```

8114|     if ( OriginalVolumeHandle !=
      | INVALID_HANDLE_VALUE ) {
8115|         ReadGood = DeviceIoControl (
8116|             OriginalVolumeHandle,
8117|             IOCTL_MOUNTDEV_QUERY_UNIQUE_ID,
8118|             NULL,
8119|             0,
8120|             UniqueRaw,
8121|             sizeof(UniqueRaw),
8122|             &dwBytesReturned,
8123|             NULL );
8124|
8125|         if ( !ReadGood ) {
8126|             Err = GetLastError();
8127|             if ( Err == ERROR_MORE_DATA ) {
8128|                 Err = 0;
8129|                 ReadGood = TRUE;
8130|             }
8131|         }
8132|
8133|         if ( ReadGood ) {
8134|             ULONG ConvertErr = 0;
8135|             WCHAR DumpString [128] = {0};
8136|             ULONG DumpStringLength =
      | sizeof(DumpString) / sizeof(DWORD);
8137|             if ( dwBytesReturned >=
      | sizeof(USHORT)+sizeof(DWORD) &&
8138|                 UniquePtr->UniqueIdLength >=
      | sizeof(DWORD) ) {
8139|                 *UniqueId = *((DWORD *)
      | UniquePtr->UniqueId);
8140|                 ConvertErr = BufferToHexWChar(
8141|                     UniquePtr->UniqueId,
8142|                     | min(dwBytesReturned,UniquePtr->UniqueIdLength),
8143|                     DumpString,
8144|                     &DumpStringLength );
8145|
8146|                 //DLOG((TEXT("GetVolumeIdentifiers:
      | UniqueId=%08x, ConvertErr=%08x, StringLen=%d,
      | String='%S'\n"),*UniqueId,ConvertErr,DumpStringLength,Du
      | mpString));
8147|             } else {
8148|                 Err = PSM_ERROR_UNSUCCESSFUL;
8149|             }
8150|         }
8151|
8152|         CloseHandle (OriginalVolumeHandle);
8153|         OriginalVolumeHandle =
      | INVALID_HANDLE_VALUE;

```

```

8154|     } else {
8155|         Err = GetLastError();
8156|         //DLOG((TEXT("GetVolumeIdentifiers:
| CreateFileW("%S") returned
| %08x\n"),OriginalVolumePath,Err));
8157|     }
8158| }
8159|
8160| return Err;
8161| }
8162|
8163| //-----
| -----
| -----
8164|
8165| DLLEXPORT PSMSTATUS PSMAPI
| Psm_DisasterRecoveryBackupSupported (void)
8166| {
8167|     return STATUS_SUCCESS;
8168| }
8169|
8170| //-----
| -----
| -----
8171|
8172| DLLEXPORT PSMSTATUS PSMAPI Psm_VolumeImageDumpW (
8173|     WCHAR                *VolumeGuid, //
| L"Volume{...}" with no trailing backslash
8174|     WCHAR
| *OriginalVolumeName,
8175|     PSM_SS_VOLUME_IMAGE_CALLBACK CallBackFunc )
8176| {
8177|     PSMSTATUS Status = 0;
8178|     PSMSTATUS TempStatus = 0;
8179|     DWORD BytesRead = 0;
8180|     BOOL ReadGood = FALSE;
8181|     const unsigned GRANULE_SIZE = 64 * 1024; // ???
| Get from driver ???
8182|     ULONG GranulesPerWorkUnit = 16*1024;
8183|     ULONG GranulesLeftInWorkUnit = 0;
8184|     ULONG SectorsPerCluster = 0;
8185|     ULONG BytesPerSector = 0;
8186|     ULONG NumberOfFreeClusters = 0;
8187|     ULONG TotalNumberOfClusters = 0;
8188|     ULONG VolumeGuidChars = wcslen(VolumeGuid);
8189|     WCHAR *VolumeGuidWithBackslash =
| LocalAlloc(LPTR,sizeof(WCHAR)*(10+VolumeGuidChars));
8190|     PARTITION_INFORMATION partitionInfo = {0};
8191|
8192|     if ( !VolumeGuidWithBackslash ) {

```

```

8193|     Status = ERROR_OUTOFMEMORY;
8194| } else {
8195|     wcsncpy ( VolumeGuidWithBackslash, VolumeGuid );
8196|     VolumeGuidWithBackslash[VolumeGuidChars++] =
| '\\';
8197|     VolumeGuidWithBackslash[VolumeGuidChars] = 0;
8198|
8199|     ReadGood = GetDiskFreeSpaceW (
8200|         VolumeGuidWithBackslash,
8201|         &SectorsPerCluster,
8202|         &BytesPerSector,
8203|         &NumberOfFreeClusters,
8204|         &TotalNumberOfClusters );
8205|
8206|     if ( !ReadGood ) {
8207|         Status = GetLastError();
8208|     } else {
8209|         ULONG ClusterSizeInBytes = BytesPerSector *
| SectorsPerCluster;
8210|         ULONG ClustersPerGranule = GRANULE_SIZE /
| ClusterSizeInBytes;
8211|         ULONG ClustersPerWorkUnit =
| GranulesPerWorkUnit * ClustersPerGranule;
8212|         ULONG VolumeBitmapSize =
| (ClustersPerWorkUnit + 7) / 8;
8213|         ULONG VolumeBitmapBufferSize =
| VolumeBitmapSize + sizeof(VOLUME_BITMAP_BUFFER);
8214|         STARTING_LCN_INPUT_BUFFER StartingLcn =
| {0};
8215|
8216|         HANDLE VolumeHandle = CreateFileW(
8217|             VolumeGuid,           // file
| name
8218|             GENERIC_READ,         //
| access mode
8219|             FILE_SHARE_READ|FILE_SHARE_WRITE,
| // share mode
8220|             NULL,                 // SD
8221|             OPEN_EXISTING,        // how
| to create
8222|             FILE_ATTRIBUTE_NORMAL,
| // file attributes
8223|             NULL );              //
| handle to template file
8224|
8225|         if ( VolumeHandle != INVALID_HANDLE_VALUE )
| {
8226|             char *GranuleBuffer =
| LocalAlloc(LPTR,GRANULE_SIZE);
8227|             VOLUME_BITMAP_BUFFER

```

```

| *VolumeBitmapBuffer =
| LocalAlloc(LPTR,VolumeBitmapBufferSize);
8228|
8229|         if ( GranuleBuffer &&
| VolumeBitmapBuffer ) {
8230|             DWORD CallBackStatus = 0;
8231|             tPSM_VolumeImageCallBackParms
| CallBackParms = {0};
8232|             ULARGE_INTEGER VolSizeInBytes =
| {0};
8233|
8234|             VolSizeInBytes.QuadPart =
8235|             ((unsigned __int64)
| TotalNumberOfClusters) *
8236|             ((unsigned __int64)
| ClusterSizeInBytes);
8237|
8238|             CallBackParms.MessageType =
| PSM_VIBMT_START;
8239|
| CallBackParms.MessageData.Start.Handle = VolumeHandle;
8240|
| CallBackParms.MessageData.Start.SizeInBytes =
| VolSizeInBytes;
8241|
| CallBackParms.MessageData.Start.ClusterSize =
| ClusterSizeInBytes;
8242|
| CallBackParms.MessageData.Start.NumUsedClusters.QuadPart
| = TotalNumberOfClusters - NumberOfFreeClusters;
8243|
8244|             Status = (*CallBackFunc)
| (&CallBackParms); // send back START chunk
8245|             if ( Status == 0 ) {
8246|                 ReadGood = DeviceIoControl (
8247|                 VolumeHandle,
8248|
| IOCTL_DISK_GET_PARTITION_INFO, // dwIoControlCode
| operation
8249|                 NULL,
| // lpInBuffer; must be NULL
8250|                 0,
| // nInBufferSize; must be zero
8251|                 &partitionInfo,
| // output buffer
8252|                 sizeof(partitionInfo),
| // size of output buffer
8253|                 &BytesRead,
| // number of bytes returned
8254|                 NULL );

```

```

| // OVERLAPPED structure
8255|
8256|         if ( !ReadGood ) {
8257|             Status = GetLastError();
8258|         } else {
8259|             Status =
| GetVolumeIdentifiers (
8260|                 OriginalVolumeName,
8261|                 VolumeHandle,
8262|                 &CallbackParms.MessageData.PartitionInfo.VolumeSerialNum
| ber,
8263|                 &CallbackParms.MessageData.PartitionInfo.VolumeUniqueid
| );
8264|
8265|             if ( Status == 0 ) {
8266|                 CallbackParms.MessageType = PSM_VIBMT_PARTITION_INFO;
8267|                 CallbackParms.MessageData.PartitionInfo.StartingOffset
| = partitionInfo.StartingOffset;
8268|                 CallbackParms.MessageData.PartitionInfo.PartitionLength
| = partitionInfo.PartitionLength;
8269|                 CallbackParms.MessageData.PartitionInfo.PartitionNumber
| = partitionInfo.PartitionNumber;
8270|                 CallbackParms.MessageData.PartitionInfo.PartitionType
| = partitionInfo.PartitionType;
8271|                 CallbackParms.MessageData.PartitionInfo.BootIndicator
| = partitionInfo.BootIndicator;
8272|                 CallbackParms.MessageData.PartitionInfo.RecognizedPartit
| ion = partitionInfo.RecognizedPartition;
8273|                 CallbackParms.MessageData.PartitionInfo.Reserved
| = 0xa5;
8274|                 Status =
| (*CallbackFunc) (&CallbackParms); // send back
| PARTITION_INFO chunk
8275|
8276|                 if ( Status == 0 ) {
8277|                     ULARGE_INTEGER
| GranuleOffset = {0};
8278|                     ULONG
| BitMapByteCount = (ClustersPerGranule + 7) / 8;
8279|

```



```

    | CallBackParms.MessageType = PSM_VIBMT_GRANULE;
8280|
    | CallBackParms.MessageData.Granule.Data = GranuleBuffer;
8281|
    | CallBackParms.MessageData.Granule.ClusterSizeInBytes =
    | ClusterSizeInBytes;
8282|
    | CallBackParms.MessageData.Granule.BitMapSizeInBytes =
    | BitMapByteCount;
8283|                while ( Status==0
    | && GranuleOffset.QuadPart<VolSizeInBytes.QuadPart ) {
8284|                ULONG
    | GranuleInUse = FALSE;
8285|                ULONG
    | BitMapByteIndex = 0;
8286|
8287|                if (
    | GranulesLeftInWorkUnit == 0 ) {
8288|
    | GranulesLeftInWorkUnit = GranulesPerWorkUnit;
8289|
    | StartingLcn.StartingLcn.QuadPart =
    | GranuleOffset.QuadPart / ClusterSizeInBytes;
8290|
8291|                ReadGood =
    | DeviceIoControl (
8292|                | VolumeHandle,
8293|                | FSCTL_GET_VOLUME_BITMAP,
8294|                | &StartingLcn,
8295|                | sizeof(StartingLcn),
8296|                | VolumeBitmapBuffer,
8297|                | VolumeBitmapBufferSize,
8298|                | &BytesRead,
8299|                NULL );
8300|
8301|                if (
    | !ReadGood ) {
8302|                ULONG
    | BitMapError = GetLastError();
8303|                if (
    | BitMapError != ERROR_MORE_DATA ) {
8304|
    | Status = BitMapError;

```

```

8305|
    | break;
8306|                }
8307|            }
8308|
8309|
    | CallbackParms.MessageData.Granule.ClusterUsedBitMap =
    | VolumeBitmapBuffer->Buffer;
8310|                }
8311|
8312|                // Figure out
    | whether any of the clusters in the current granule
8313|                // are in use.
    | If so, the entire granule is in use and must be
8314|                // read and
    | sent to the callback routine.
8315|                // Otherwise,
    | we can skip the granule.
8316|
8317|                for (
    | BitMapByteIndex=0; BitMapByteIndex < BitMapByteCount;
    | ++BitMapByteIndex ) {
8318|                    if (
    | CallbackParms.MessageData.Granule.ClusterUsedBitMap[BitM
    | apByteIndex] ) {
8319|
    | GranuleInUse = TRUE;
8320|                    break;
8321|                }
8322|            }
8323|
8324|            if (
    | GranuleInUse ) {
8325|                ReadGood =
    | ReadFile (
8326|
    | VolumeHandle,
8327|
    | GranuleBuffer,
8328|
    | GRANULE_SIZE,
8329|
    | &BytesRead,
8330|                NULL );
8331|
8332|            if (
    | !ReadGood ) {
8333|                Status
    | = GetLastError();
8334|                break;

```

```

8335|             }
8336|
8337|         | CallbackParms.MessageData.Granule.GranuleSizeInBytes =
         | BytesRead;
8338|         | CallbackParms.MessageData.Granule.GranuleOffsetInBytes.Q
         | uadPart = GranuleOffset.QuadPart;
8339|
8340|             Status =
         | (*CallbackFunc) (&CallbackParms);
8341|
8342|             if (
         | BytesRead < GRANULE_SIZE ) {
8343|                 break;
8344|             }
8345|         } else {
8346|
         | LARGE_INTEGER DistanceToMove;
8347|         | DistanceToMove.QuadPart = GRANULE_SIZE;
8348|             ReadGood =
         | SetFilePointerEx (
8349|         | VolumeHandle,
8350|         | DistanceToMove,
8351|             NULL,
8352|         | FILE_CURRENT );
8353|
8354|             if (
         | !ReadGood ) {
8355|                 Status
         | = GetLastError();
8356|                 break;
8357|             }
8358|         }
8359|
8360|         | GranuleOffset.QuadPart += GRANULE_SIZE;
8361|         | --GranulesLeftInWorkUnit;
8362|         | CallbackParms.MessageData.Granule.ClusterUsedBitMap +=
         | ClustersPerGranule / 8;
8363|     }
8364|
8365|     | CallbackParms.MessageType = PSM_VIBMT_FINISH;

```

```

8366|
8367|     | CallbackParms.MessageData.Finish.Status = Status;
8367|         TempStatus =
8368|     | (*CallbackFunc) (&CallbackParms);
8368|         if ( Status == 0 )
8369|     | {
8369|         Status =
8370|     | TempStatus;
8370|     }
8371|     }
8372|     }
8373|     }
8374|     }
8375|     } else {
8376|         Status = ERROR_OUTOFMEMORY;
8377|     }
8378|
8379|     if ( GranuleBuffer ) {
8380|         LocalFree(GranuleBuffer);
8381|         GranuleBuffer = NULL;
8382|     }
8383|
8384|     if ( VolumeBitmapBuffer ) {
8385|         LocalFree(VolumeBitmapBuffer);
8386|         VolumeBitmapBuffer = NULL;
8387|     }
8388|
8389|     CloseHandle (VolumeHandle);
8390|     VolumeHandle = INVALID_HANDLE_VALUE;
8391| } else {
8392|     Status = GetLastError();
8393| }
8394| }
8395|
8396| LocalFree(VolumeGuidWithBackslash);
8397| VolumeGuidWithBackslash = NULL;
8398| }
8399|
8400| return Status;
8401| }
8402|
8403| //-----
8404| | -----
8405| STATIC BOOL PerThreadInit (
8406|     HINSTANCE hInstance,
8407|     DWORD fdwReason )
8408| {
8409|     BOOL bResult = TRUE;
8410|     pthreadStorage ThreadStorage = (pthreadStorage)

```

```

    | LocalAlloc(LPTR,sizeof(tThreadStorage));
8411|
8412|     if(!ThreadStorage) {
8413|         bResult = FALSE;
8414|     } else {
8415|         InitThreadLocalStorage (ThreadStorage);
8416|         ThreadStorage->hInstance = hInstance;
8417|         ThreadStorage->fdwReason = fdwReason;
8418|         TlsSetValue(TlsIndex,ThreadStorage);
8419|     }
8420|
8421|     return bResult;
8422| }
8423|
8424| //-----
    | -----
8425|
8426| STATIC BOOL PerThreadCleanup()
8427| {
8428|     BOOL bResult = TRUE;
8429|     pThreadStorage ThreadStorage = GetThreadStorage();
8430|
8431|     if ( ThreadStorage ) {
8432|         LocalFree((HLOCAL) ThreadStorage);
8433|     }
8434|
8435|     return bResult;
8436| }
8437|
8438| //-----
    | -----
8439|
8440| BOOL WINAPI DllMain (
8441|     HINSTANCE hDllInst,
8442|     DWORD     fdwReason,
8443|     LPVOID    lpvReserved )
8444| {
8445|     BOOL bResult = TRUE;
8446|
8447|     switch (fdwReason) {
8448|         case DLL_PROCESS_ATTACH: {
8449|             // The DLL is being loaded for the first
            | time by a given process.
8450|             // Perform per-process initialization here.
            | If the initialization
8451|             // is successful, return TRUE; if
            | unsuccessful, return FALSE.
8452|             TlsIndex = TlsAlloc();
8453|             if(TlsIndex==0xffffffff) {
8454|                 bResult = FALSE;

```

```

8455|         } else {
8456|             bResult = PerThreadInit (hDllInst,
8457| | fdwReason);
8458|         }
8459|     } break;
8460|     case DLL_THREAD_ATTACH: {
8461|         // A thread is being created in a process
8462|         | that has already loaded
8463|         // this DLL. Perform any per-thread
8464|         | initialization here. The
8465|         // return value is ignored.
8466|         bResult = PerThreadInit (hDllInst,
8467| | fdwReason);
8468|     } break;
8469|     case DLL_PROCESS_DETACH: {
8470|         bResult = PerThreadCleanup();
8471|         TlsFree(TlsIndex);
8472|         TlsIndex = 0xffffffff;
8473|     } break;
8474|     case DLL_THREAD_DETACH: {
8475|         // A thread is exiting cleanly in a process
8476|         | that has already
8477|         // loaded this DLL. Perform any per-thread
8478|         | clean up here. The
8479|         // return value is ignored.
8480|         bResult = PerThreadCleanup();
8481|     } break;
8482|     default: {
8483|         // What in tarnation is going on around
8484|         | here?
8485|         bResult = FALSE;
8486|     } break;
8487| }
8488| return bResult;
8489| }
8490|
8491|
8492|
8493| File Listing: drbackup.h
8494|
8495| /*
8496| //-----
8497| | -----

```

```

| -----
8497| // The following function returns STATUS_SUCCESS if
| DR Backup is supported by this version of drbackup.dll.
8498| // It returns PSM_ERROR_NOT_IMPLEMENTED if DR Backup
| is NOT supported.
8499| //
8500| PSMSTATUS PSMAPI Psm_DisasterRecoveryBackupSupported
| (void);
8501|
8502|
8503| //-----
| -----
| -----
8504| // The following function is the engine that drives a
| volume image traversal.
8505| // VolumeGuid is the device name of a snapshot.
| Actually, it can be live volume, but that's silly!
8506| // OriginalVolumeName is the name of the live volume
| that the snapshot is of.
8507| // CallBackFunc is a pointer to a function that
| handles all the backup image events.
8508| // If it returns anything other than STATUS_SUCCESS,
| the backup is immediately aborted.
8509| //
8510| PSMSTATUS PSMAPI Psm_VolumeImageDumpW (
8511|     WCHAR                *VolumeGuid,
| // L"Volume{...}" with no trailing backslash
8512|     WCHAR
| *OriginalVolumeName, // Whether or not VolumeGuid is
| a snapshot, this needs to be original volume
8513|     PSM_SS_VOLUME_IMAGE_CALLBACK CallBackFunc );
8514|
8515| /*--- end of file drbackup.h ---*/
8516|
8517|
8518|
8519| PSM Disaster Recovery Boot System Source
8520|
8521| The Recovery Boot disk is a single floppy that when
| booted will attach to a server or local drive
| containing a corresponding backup. --LPW
8522|
8523|
8524|
8525| File Listing: AUTOEXEC.bat
8526|
8527| @echo Off
8528| set DEBUG=
8529| set WPWAIT=WPWAIT
8530| set WEWAIT=WEWAIT

```

```

8531| set OKWAIT=OKWAIT
8532| IF "%DEBUG%"==" " SET ToNul=Nul
8533| IF "%DEBUG%"==" " SET ToOff=Off
8534| IF NOT "%DEBUG%"==" " SET ToNul=Con
8535| IF NOT "%DEBUG%"==" " SET ToOff=On
8536| @echo %ToOff%
8537| set LOGFILE=NUL
8538| if exist A:\APP\APP_LOGO.BAT call A:\APP\APP_LOGO
    | AUTOEXEC.BAT
8539| echo *** Building RamDisk
8540| call a:\dos\setramd.bat
8541| %RAMD%
8542| path=%RAMD%\APP;%RAMD%\;%RAMD%\DOS;%RAMD%\NET;%RAMD%\INI
    | ;
8543| prompt $p$g
8544| copy A:\command.com %RAMD%\ >%ToNul%
8545| set comspec=%RAMD%\command.com
8546| copy a:*.bat >%ToNul%
8547| copy a:\fixboot.exe >%ToNul%
8548| call %ramd%\wp_wait.bat
8549| call %ramd%\startup.bat
8550| %ramd%
8551| CD \
8552| call %ramd%\RESULT.BAT
8553| if exist %ramd%\APP_DOES.BAT call %ramd%\APP_DOES.BAT
8554|
8555| %ramd%
8556|
8557| echo Update Log File(s)
8558| if exist %RAMD%\*.LOG goto relog
8559| goto fin
8560| :relog
8561| rem Check for Write Enable
8562| call %ramd%\we_wait
8563| copy a:\head.txt + %RAMD%\*.LOG + a:\end.txt
    | a:\results.htm /Y
8564| :fin
8565| %ramd%\ok_WAIT.bat
8566| :end
8567|
8568|
8569|
8570| File Listing: CHANGE.bas
8571|
8572| 'CHANGE.BAS
8573| '
8574| 'Function:
8575| ' Does global search and replace on a file (wildcards
    | are okay)
8576| ' has ability to handle special batch file characters,

```



```

| deal
8577| ' with quoted strings, and embedded double quotes.
8578| '
8579| ' Handy for revising BAT and INI files on a grand
| scale
8580| ' Originally developed to setup PROTOCOL.INI AND
| SYSTEM.INI
8581| ' for a DOS boot disk.
8582| '
8583| ' Very fast and smart
8584| ' Does in-place update to keep ver control happy
8585| '
8586| 'Bugs:
8587| ' Won't process files longer than 31K
8588| ' Very little error checking
8589| '
8590| ' run with /? to see instructions
8591| '
8592| 'Make:
8593| ' Get PowerBasic 3.5 for DOS from www.powerbasic.com
8594| ' Get PKLite for DOS from www.pkware.com
8595| 'Use PB UI to compile change.bas -> change.exe
8596| 'Use PKLite to compress change.exe -> gsr.exe
8597| '
8598| 'History:
8599| ' 04/13/2001 Lou Witt:
8600| ' Original Work
8601| '
8602| '
8603| defint a-z
8604| declare sub qtrim(a$)
8605| declare sub parse_args( cmd$, param$(), paramcount,
| pparamlimit)
8606| declare sub SayHelp()
8607| declare sub GSR
| (WorkString$,Search$,Repl$,NoCase,Hits,Changes)
8608| declare function YesNo$(Flag)
8609| '
8610| %paramlimit=15
8611| dim param$(%paramlimit):paramcount = 0:
8612| flags$ = ""
8613| fixcase = -1 'case insensitive flag
8614| fixbat = -1 'batch mode flag
8615| verbose = 0 'noisy
8616| FileCount = 1
8617| dim dynamic WorkFiles$(FileCount)
8618| handle = 1
8619| FileSizeLimit = 32000
8620| Quote$ = chr$(34)
8621| Really = -1

```

```

8622|
8623|
8624| parse_args command$, param$(), paramcount, %paramlimit
8625|
8626| 'help? or null?
8627| if (paramcount=0) then
8628|   paramcount = 1
8629|   param$(1) = "/H"
8630| end if
8631|
8632| 'parse flags
8633| aparam$ = param$(1)
8634| do while instr("/-",left$(aparam$,1))
8635|   'do the flag
8636|   flags$ = flags$ + lcase$(mid$(aparam$,2,1))
8637|   if paramcount>1 then
8638|     for n = 2 to paramcount
8639|       param$(n-1) = param$(n)
8640|     next
8641|     aparam$ = param$(1)
8642|   else
8643|     aparam$="x"
8644|   end if
8645|   param$(paramcount)=""
8646|   decr paramcount
8647| loop
8648|
8649|
8650| if instr(Flags$, Any "h?") then
8651|   SayHelp
8652| end
8653| end if
8654|
8655| if instr(Flags$, Any "!") then
8656|   replace "!" with "" in Flags$
8657|   fixcase = 0
8658| end if
8659|
8660| if instr(Flags$, Any "=") then
8661|   replace "=" with "" in Flags$
8662|   fixbat = 0
8663| end if
8664|
8665| if instr(Flags$, Any "v") then
8666|   replace "v" with "" in Flags$
8667|   verbose = -1
8668| end if
8669|
8670| if Flags$<>"" then
8671|   print "Unexpected Argument " ;

```

```

8672| print flags$;
8673| print ""
8674| SayHelp
8675| end
8676| end if
8677|
8678| if paramcount=2 then
8679|     Really = 0
8680| end if
8681|
8682| if (paramcount <> 3) and (paramcount <> 2) then
8683|     Print "Bad Parameter Count"
8684|     SayHelp
8685|     end
8686| end if
8687|
8688| FileMask$ = Param$(1)
8689| Search$ = Param$(2)
8690| Repl$ = Param$(3)
8691|
8692| if FixBat then
8693|     ' Make it command line compatible
8694|     'NOT replace {}~@ with <|>&% in search and replace
8695|     replace "{" with "<" in Search$
8696|     replace "!" with "|" in Search$
8697|     replace "}" with ">" in Search$
8698|     replace "~" with "&" in Search$
8699|     replace "" with "%" in Search$
8700|     replace "{" with "<" in Repl$
8701|     replace "!" with "|" in Repl$
8702|     replace "}" with ">" in Repl$
8703|     replace "~" with "&" in Repl$
8704|     replace "" with "%" in Repl$
8705| end if
8706|
8707| ' *** Split out the path ***
8708| WorkPath$=""
8709| if instr(FileMask$,any "\:")>0 then
8710|     ' has some kind of path
8711|     mp = len(FileMask$)
8712|     do
8713|         cp = instr(mp,FileMask$,any "\:")
8714|         decr mp
8715|     loop until mp=0 or cp>0
8716|     WorkPath$ = left$(FileMask$,cp)
8717| end if
8718|
8719| if verbose then
8720|     ? "Files: ";Quote$;FileMask$;Quote$
8721|     ? "Search: ";Quote$;Search$;Quote$

```

```

8722|   ? "Replace:",Quote$;Repl$;Quote$
8723|   ? "Batch Xlate:",YesNo$(FixBat)
8724|   ? "Any Case:",YesNo$(FixCase)
8725| end if
8726|
8727| ' *** Get the file list ***
8728| WorkFile$ = Dir$(FileMask$)
8729| if WorkFile$ = "" then
8730|   End
8731| end if
8732|
8733| Do While WorkFile$ <> ""
8734|   WorkFiles$(FileCount) = WorkFile$
8735|   WorkFile$ = Dir$
8736|   if WorkFile$<> "" then
8737|     Incr FileCount
8738|     Redim preserve WorkFiles$(FileCount)
8739|   end if
8740| Loop
8741|
8742|
8743| ' *** Work the File List ***
8744| For FileNum = 1 to FileCount 'and files?
8745|   'for this file
8746|   WorkFile$ = WorkFiles$(FileNum)
8747|   if verbose then
8748|     ? "File:",WorkFile$,
8749|   end if
8750|   Open WorkPath$ & WorkFile$ for Binary as Handle
8751|   Hits = 0: Changes = 0
8752|   FileLen = Lof(Handle)
8753|   if (FileLen > 0) and (FileLen < FileSizeLimit) then
8754|     Get$ Handle, FileLen, FileBody$
8755|     GSR FileBody$,Search$,Repl$,FixCase,Hits,Changes
8756|   end if
8757|   if Hits<> 0 then
8758|     if Really then
8759|       'write the update
8760|       Seek Handle,0
8761|       Put$ Handle, FileBody$
8762|       SetEof Handle
8763|     end if
8764|   end if
8765|   Close Handle
8766|   if verbose then
8767|     ? "Hits: ";Hits;
8768|     if really then
8769|       print ,"Changes: ";Changes
8770|     else
8771|       print

```

```

8772|     end if
8773| end if
8774| next
8775| if verbose then
8776|     print "Job Done"
8777| end if
8778| end
8779|
8780| *** Application Subroutines ***
8781| function YesNo$(Flag)
8782|     if Flag <> 0 then
8783|         YesNo$ = "Yes"
8784|     else
8785|         YesNo$ = "No"
8786|     end if
8787| end function
8788|
8789| sub SayHelp
8790|     'show help
8791|     print "Change [!]/[h]/[/]/[v] file search
      | replace"
8792|     print" file filename or wildcard mask"
8793|     print" /! causes case SENSITIVE search"
8794|     print" /= to NOT replace {!}~` with <|>&% in search
      | and replace"
8795|     print" /?./H displays this help"
8796|     print" /v Verbose output"
8797|     print" Search and Replace may be quoted"
8798|     print" To include or embed quotes, double them"
8799| end sub
8800|
8801|
8802| sub GSR
      | (WorkString$,Search$,Repl$,FixCase,Hits,Changes)
8803| Hits = 0
8804| Changes = 0
8805| RLen = Len(Repl$)
8806| if len(WorkString$)=0 then Exit Sub
8807| SLen = Len(Search$)
8808| if len(Search$)=0 then Exit Sub
8809| if 0=FixCase then
8810|     'this is fast way
8811|     replace Search$ with Repl$ in WorkString$
8812|     exit sub
8813| else
8814|     'have to do it the slowwww way
8815|     cx$ = LCase$(WorkString$)
8816|     s$ = LCase$(Search$)
8817|     pad$ = String$(RLen,255)
8818|     cp = instr(cx$,s$)

```

```

8819| do while cp>0
8820| 'we have a hit
8821|     Incr Hits
8822|     ' clean the mask
8823|     cx$ = left$(cx$,cp-1) & Pad$ &
        | mid$(cx$,cp+Slen)
8824|     ' if redundant, skip it
8825|     Hit = -1
8826|     if Slen = Rlen then
8827|         Hit = Repl$ <> Mid$(WorkString$,Cp,Rlen)
8828|     end if
8829|     if Hit then
8830|         WorkString$ = left$(WorkString$,cp-1) & Repl$ &
            | Mid$(WorkString$,cp+Slen)
8831|         Incr Changes
8832|     end if
8833|     cp = instr(cx$,s$)
8834| loop
8835| end if
8836| end sub
8837|
8838|
8839|
8840| ***** Library Routines *****
8841| sub parse_args( cmd$, pparam$(), pparamcount,
        | pparamlimit)
8842|
8843| *** unquote command$ to cmx$
8844| cmx$ = lcase$(cmd$) + " "
8845| iq = 0
8846| aq = 0
8847| for cp = 1 to len(cmd$)
8848|     ch = asc(cmx$,cp)
8849|     aq = ch = 34
8850|     if aq then
8851|         iq = not iq
8852|         ch = 95
8853|     end if
8854|     if iq then
8855|         asc(cmx$,cp) = 95
8856|     else
8857|         asc(cmx$,cp) = ch
8858|     end if
8859| next
8860|
8861| *** parse multiple arguments
8862|
8863| cp = 1
8864| inarg = 0
8865| anarg$ = ""

```

```

8866| aspace = 0
8867| do while cp <= len(cmx$)
8868|   ch = asc(cmx$,cp)
8869|   cmm$ = space$(cp-1)+"^"
8870|   aspace = (ch = 32) or (ch = 9)
8871|   if inarg then
8872|     if aspace then
8873|       incr pparamcount
8874|       if pparamcount <= pparamlimit then
8875|         qtrim anarg$
8876|         pparam$(pparamcount) = anarg$
8877|       end if
8878|       inarg = 0
8879|     else
8880|       anarg$ = anarg$ + mid$(cmd$,cp,1)
8881|     end if
8882|   else
8883|     if not aspace then
8884|       inarg = -1
8885|       anarg$ = mid$(cmd$,cp,1)
8886|     end if
8887|   end if
8888|   incr cp
8889| loop
8890| if inarg then
8891|   incr pparamcount
8892|   if pparamcount < pparamlimit then
8893|     qtrim anarg$
8894|     pparam$(pparamcount) = anarg$
8895|   end if
8896| end if
8897| pparamcount = min(pparamlimit,pparamcount)
8898| end sub
8899|
8900| sub qtrim(a$)
8901| 'trim a string of leading/trailing quotes, singulate
   | doubles
8902| if (asc(a$,1) = 34) and (asc(a$,len(a$)) = 34) then
8903|   a$ = mid$(a$,2)
8904|   a$ = left$(a$,len(a$)-1)
8905| end if
8906| cm$ = chr$(34) + chr$(34)
8907| cn = 1
8908| cp = instr(cn,a$,cm$)
8909| do while cp<>0
8910|   a$ = left$(a$,cp) + mid$(a$,cp+2)
8911|   cn = cp + 1
8912|   cp = instr(cn,a$,cm$)
8913| loop
8914| end sub

```

```
8915|
8916|
8917|
8918| File Listing: CONNECT.bat
8919|
8920| rem @echo %ToOff% $
8921| REM *** Attempt three shares
8922| echo %user% >up.txt
8923| echo %password% >>up.txt
8924| type up.txt|net use /Y
8925| REM Try1
8926| if ""=="%SERVER1%" goto Try2
8927| echo *** Connecting %DRIVE1% TO %SERVER1%\%SHARE1%
8928| echo *** Connecting %DRIVE1% TO
      | %SERVER1%\%SHARE1%>>%LOGFILE%
8929| net use %DRIVE1% \\%SERVER1%\%SHARE1% %password%
      | /SAVEPW:NO /PERSISTENT:NO /YES
8930| :Try2
8931| if ""=="%SERVER2%" goto Try3
8932| echo *** Connecting %DRIVE2% TO %SERVER2%\%SHARE2%
8933| echo *** Connecting %DRIVE2% TO
      | %SERVER2%\%SHARE2%>>%LOGFILE%
8934| net use %DRIVE2% \\%SERVER2%\%SHARE2% %password%
      | /SAVEPW:NO /PERSISTENT:NO /YES
8935| :Try3
8936| if ""=="%SERVER3%" goto DoApp
8937| echo *** Connecting %DRIVE3% TO %SERVER3%\%SHARE3%
8938| echo *** Connecting %DRIVE3% TO
      | %SERVER3%\%SHARE3%>>%LOGFILE%
8939| net use %DRIVE3% \\%SERVER3%\%SHARE3% %password%
      | /SAVEPW:NO /PERSISTENT:NO /YES
8940| REM Let the App check for errors
8941| :DoApp
8942| :END
8943|
8944|
8945|
8946| File Listing: CRS.bat
8947|
8948| This file contains three carriage returns --LPW
8949|
8950|
8951|
8952| File Listing: FATAL.bat
8953|
8954| rem @echo %ToOff% $
8955| SET RESULT=%1%2%3%4%5%6%7%8%9
8956| echo A Fatal Error Occured (%RESULT%)
8957| echo A Fatal Error Occured (%RESULT%) >>%LOGFILE%
8958| rem Redefine the result value for calling shell
```



```

8959| echo SET RESULT=%RESULT% >RESULT.BAT
8960| EXIT
8961|
8962|
8963|
8964| File Listing: MAKEDISK.bas
8965|
8966| ' PowerBASIC 3.2 source code for formatting floppy
    | disks
8967| ' Original by Thaddy De Konig
8968| ' Additions and modifications by Dave Navarro, Jr.
    | (dave@powerbasic.com)
8969| ' Revised to run silent w/o format by Lou Witt
    | (lwitt@cdp.com) 4/17/2001
8970| ' Post Proc: PKLite -c -o MakeDisk.exe MKD.Exe
8971| ' Function: Writes a Win95/98 boot sector to floppy
    | in A:
8972| ' Revised for retry, message, error return LouWitt
    | 8/23/01
8973| '
8974| '=====
    | =====
8975|
8976| $DIM ALL
8977|
8978| DECLARE FUNCTION PBFORMAT(BYVAL drive AS INTEGER, BYVAL
    | media AS INTEGER) AS INTEGER
8979|
8980| DECLARE FUNCTION ValidateDisk(BYVAL drive AS INTEGER,
    | BYVAL media AS INTEGER) AS INTEGER
8981| DECLARE FUNCTION FormatDisk(BYVAL drive AS INTEGER) AS
    | INTEGER
8982| DECLARE FUNCTION WriteBoot(BYVAL drive AS INTEGER) AS
    | INTEGER
8983| DECLARE FUNCTION WriteFAT(BYVAL drive AS INTEGER) AS
    | INTEGER
8984| DECLARE FUNCTION WriteDir(BYVAL drive AS INTEGER) AS
    | INTEGER
8985| DECLARE SUB InitFormatParms(BYVAL media AS INTEGER)
8986| DECLARE SUB ResetFDC(BYVAL drive AS INTEGER)
8987| DECLARE FUNCTION FormatTrack(BYVAL drive AS INTEGER,
    | BYVAL track AS INTEGER, BYVAL head AS INTEGER) AS
    | INTEGER
8988| DECLARE FUNCTION WriteBootSector(BYVAL drive AS
    | INTEGER) AS INTEGER
8989| DECLARE SUB ComputeCHS( LogSec AS INTEGER, cyl AS
    | INTEGER, hd AS INTEGER, sec AS INTEGER)
8990| DECLARE FUNCTION WriteSector( BYVAL drive AS INTEGER,
    | BYVAL cylinder AS INTEGER, BYVAL head AS INTEGER, BYVAL
    | sector AS INTEGER, BYVAL Buffer AS DWORD) AS INTEGER

```

```

8991|
8992|
8993| %True = -1
8994| %False = 0
8995|
8996|
8997| %FLAGS = 0
8998| %AX   = 1
8999| %BX   = 2
9000| %CX   = 3
9001| %DX   = 4
9002| %SI   = 5
9003| %DI   = 6
9004| %BP   = 7
9005| %DS   = 8
9006| %ES   = 9
9007|
9008| %F360 = &HFD
9009| %F1200 = &HF9
9010| %F720 = - &HF9 ' media byte is NEGATIVE to differ from
    | %F1200
9011| %F1440 = &HF0
9012| %F2880 = - &HF0 ' media byte is negative to differ from
    | %F1440
9013|
9014| %RETRIES = 5 '%RETRIES on BIOS error
9015|
9016| TYPE ADDRFIELDtype
9017| track AS STRING * 1
9018| head AS STRING * 1
9019| sector AS STRING * 1
9020| bytesec AS STRING * 1
9021| END TYPE '4
9022|
9023| TYPE INFOtype
9024| OEM AS STRING * 8 'system name
9025| BS AS WORD 'bytes/sector
9026| SC AS BYTE 'STRING * 1 'sectors/cluster
9027| RS AS WORD 'reserved sectors
9028| NF AS BYTE 'STRING * 1 'FATs
9029| DE AS WORD 'root directory entries
9030| TS AS WORD 'total sectors on volume
9031| MB AS STRING * 1 'media byte
9032| SF AS WORD 'sectors/FAT
9033| ST AS WORD 'sectors/track
9034| NH AS WORD 'heads
9035| HS AS WORD 'hidden sectors
9036| END TYPE '27
9037|
9038| TYPE BOOTRECType

```

```

9039| jmp AS STRING * 3
9040| parms AS INFOtype
9041| code AS STRING * 482
9042| END TYPE '512
9043|
9044|
9045| '
9046| dim SAt as integer
9047| dim SVal as integer
9048| dim SStr as string
9049| DIM ECount as WORD
9050| DIM Verbose as integer
9051| dim prompt as integer
9052| dim TheKey as string
9053| dim HelpMe as integer
9054| dim Melody as integer
9055|
9056| 'INT 1Eh disk parameter table vectors
9057| DIM OldDPTseg AS SHARED WORD
9058| DIM OldDPToff AS SHARED WORD
9059| DIM NewDPTseg AS SHARED WORD
9060| DIM NewDPToff AS SHARED WORD
9061|
9062| 'Number of tracks on media
9063| DIM NoTracks AS SHARED INTEGER
9064|
9065| 'format info for media
9066| DIM Info AS SHARED INFOtype
9067|
9068| 'interface with INTERRUPTX routine
9069| 'boot record buffer
9070| DIM BootRec AS SHARED BOOTRECtype
9071|
9072| 'sector buffer to write FAT & root directory sectors
9073| DIM SectorBuff AS SHARED STRING * 512
9074|
9075| DIM Drive AS SHARED INTEGER
9076| DIM Media AS SHARED INTEGER
9077| DIM verifydisk AS SHARED INTEGER
9078|
9079| $STATIC
9080|
9081| 'Allocate address field data to max possible sectors
    | per track
9082| DIM AddrField( 1 TO 36 ) AS SHARED ADDRFIELDtype
9083| '=====
    | =====
9084|
9085|
9086| DIM xerr AS INTEGER

```

```

9087| DIM errl AS INTEGER
9088| DIM errc AS INTEGER
9089|
9090|
9091|
9092|
9093|
9094|
9095|
9096| Verbose = %False
9097| if instr(Command$,Any "vV") then Verbose = %True
9098| Prompt = %True
9099| if instr(Command$,Any "qQ") then Prompt = %False
9100| HelpMe = %False
9101| if instr(Command$,Any "?hH") then HelpMe = %True
9102|
9103| if HelpMe Then
9104|   print "FixBoot (c)2001 Columbia Data Products"
9105|   print "
9106|   print "Args:"
9107|   print "  /? /H Display this help"
9108|   print "  /+ /- Play a scale up or down"
9109|   print "  /s### Delay ### seconds"
9110|   print "  /q /Q Run Quiet"
9111|   print "  /v /V Run with Verbose messages"
9112|   print "  /qv Don't prompt, but display errors"
9113|   print
9114|   print "Returns:"
9115|   print "  0:no error
9116|   print "  1:invalid function request
9117|   print "  2:address mark not found
9118|   print "  3:write protected
9119|   print "  4:sector not found
9120|   print "  6:diskette changed
9121|   print "  8:DMA overrun
9122|   print "  9:DMA boundary error
9123|   print " 12:media type not available
9124|   print " 16:bad CRC
9125|   print " 32:diskette controller failed
9126|   print " 64:seek failed
9127|   print "128:time-out/drive not ready
9128|   print " 99:operator abort
9129|   print
9130|   end 0
9131| end if
9132|
9133| 'check for sound work
9134| Melody = %false
9135| if instr(Command$, "/+") then Melody= %True
9136| if instr(Command$, "-+") then Melody= %True

```

```

9137| if Melody then
9138|   gosub PlayUp
9139|   end 0
9140| end if
9141|
9142| Melody = %false
9143| if instr(Command$, "/-") then Melody= %True
9144| if instr(Command$, "--") then Melody= %True
9145| if Melody then
9146|   gosub PlayDown
9147|   end 0
9148| end if
9149|
9150| 'check for delay work
9151| if instr(Command$,ANY "sS") then
9152|   Sat = instr(Command$, ANY "sS") + 1
9153|   SStr= trim$(Mid$(Command$,Sat))
9154|   if Len(SStr)>0 then
9155|     SVal = Val(SStr)
9156|     Sleep SVal
9157|   end if
9158|   end 0
9159| end if
9160|
9161| Drive = 0
9162| media = %F1440
9163|
9164|
9165| do while inkey$<>""
9166|   sleep 0.1
9167| loop
9168|
9169| if prompt then
9170|   PRINT "Insert 1.44 disk to make bootable in drive
    | A: and"
9171|   PRINT "Press ENTER to continue...";
9172|   TheKey$ = ""
9173|   Do
9174|     TheKey$ = Inkey$
9175|   Loop Until TheKey$ <> ""
9176|   print
9177|   if asc(TheKey$) <> 13 then
9178|     print "Aborted. <";
9179|     print asc(TheKey$);">"
9180|     sleep 15
9181|     end 99
9182|   end if
9183|   print
9184|   PRINT "Writing boot sector..."
9185|   PRINT

```

```

9186| end if
9187|
9188|
9189| ECount = 5 'we will several times
9190|
9191| do while ECount > 0
9192| xerr = PBFORMAT( drive, media )
9193|   IF xerr THEN
9194|     errl = xerr \ 256
9195|     errc = xerr AND 255
9196|     if Verbose then PRINT "*** Error level: "; errl;
       | " code: "; errc
9197|     if errc = 3 then
9198|       ECount = 1 'force error exit
9199|     end if
9200|   ELSE
9201|     exit loop
9202|   END IF
9203|   Decr ECount
9204| loop
9205| If ECount <> 0 then
9206|   'we succeeded without error
9207|   if prompt then PRINT "done.":sleep 15
9208| end 0
9209| end if
9210|
9211|
9212| If Verbose or Prompt then
9213|
9214|   'display error level
9215|   Select Case errl 'level error codes
9216|     Case 0: '0:no error
9217|     Case 1: ?"1:validate error
9218|     Case 2: ?"2:format error
9219|     Case 3: ?"3:boot record write error
9220|     Case 4: ?"4:FAT write error
9221|     Case 5: ?"5:directory write error
9222|     Case Else: ?"?:Unknown error level
9223|   End Select
9224|
9225|   'display floppy error code
9226|   Select Case errc 'floppy disk error codes (xerr,in
       | decimal):
9227|     Case 0'0:no error
9228|     Case 1: ?"1:invalid function request
9229|     Case 2: ?"2:address mark not found
9230|     Case 3: ?"3:write protected
9231|     Case 4: ?"4:sector not found
9232|     Case 6: ?"6:diskette changed
9233|     Case 8: ?"8:DMA overrun

```

```

9234|    Case 9: ?""9:DMA boundary error
9235|    Case 12: ?""12:media type not available
9236|    Case 16: ?""16:bad CRC
9237|    Case 32: ?""32:diskette controller failed
9238|    Case 64: ?""64:seek failed
9239|    Case 128: ?""128:time-out/drive not ready
9240|    Case Else: ?""?:Unknown floppy error
9241| End Select
9242| end if
9243| '
9244|
9245| if prompt then
9246|   print "FixBoot Failed."
9247|   sleep 15
9248|   TheKey$ = inkey$
9249| end if
9250| end errc
9251| END
9252|
9253| PlayUp:
9254|   Play "O3 L8 C D E F G A B O4 C L4 CCCCCC"
9255|   RETURN
9256|
9257| PlayDown:
9258|   Play "O3 L8 C O2 B A G F E D C L4 CCCCCCCC"
9259|   RETURN
9260|
9261|
9262| 'level error codes
9263| '0:no error
9264| '1:validate error
9265| '2:format error
9266| '3:boot record write error
9267| '4:FAT write error
9268| '5:directory write error
9269|
9270| '
9271| FUNCTION PBFORMAT(BYVAL drive AS INTEGER, _
9272|                  BYVAL media AS INTEGER) PUBLIC AS
  | INTEGER
9273|
9274|   DIM xerr AS INTEGER
9275|   DIM level AS INTEGER
9276|
9277|   xerr = ValidateDisk( drive, media )
9278|   goto skip10
9279|   IF xerr = 0 THEN
9280|     xerr = FormatDisk( drive )
9281|   ELSE
9282|     level = 1

```

```

9283| END IF
9284| skip10:
9285| IF xerr = 0 THEN
9286|   xerr = WriteBoot( drive )
9287| ELSEIF level = 0 THEN
9288|   level = 2
9289| END IF
9290| goto skip20
9291| IF xerr = 0 THEN
9292|   PRINT "Creating FAT tables..."
9293|   xerr = WriteFAT( drive )
9294| ELSEIF level = 0 THEN
9295|   level = 3
9296| END IF
9297| IF xerr = 0 THEN
9298|   xerr = WriteDir( drive )
9299| ELSEIF level = 0 THEN
9300|   level = 4
9301| END IF
9302| IF xerr THEN
9303|   IF level = 0 THEN
9304|     level = 5
9305|   END IF
9306| END IF
9307| skip20:
9308| 'reset INT 1Eh vector to original,check both for <>0
9309| IF OldDPTseg < > 0 OR OldDPToff < > 0 THEN
9310|   REG %ax, &H251E
9311|   REG %ds, OldDPTseg
9312|   REG %dx, OldDPToff
9313|   CALL INTERRUPT &H21
9314| END IF
9315| FUNCTION = ( level * 256 ) + xerr
9316| END FUNCTION
9317|
9318| 'check if media supported by program, by drive, and if
| drive is ready
9319| FUNCTION ValidateDisk(BYVAL drive AS INTEGER, _
9320|   BYVAL media AS INTEGER) PRIVATE
| AS INTEGER
9321|
9322| DIM CMOS AS INTEGER
9323| DIM checks AS INTEGER
9324| DIM zerr AS INTEGER
9325|
9326| InitFormatParms media
9327| IF Info.ST THEN
9328| 'check to see if there is CMOS RAM (means we have an
| AT BIOS)
9329|   OUT &H70, &H10

```



```

9330|  CMOS = ( INP( &H71 ) < > &HFF )
9331|  ResetFDC drive
9332|  IF CMOS THEN
9333|  'get original INT 1Eh vector to disk parameter
    | table
9334|    REG %ax, &H351E
9335|    CALL INTERRUPT &H21
9336|    OldDPTseg = REG( %es )
9337|    OldDPToff = REG( %bx )
9338|  'set media type for format on AT BIOS/multi-media
    | drive
9339|  'let BIOS determine if drive can format media with
    | a DPT in BIOS ROM
9340|    FOR checks = 1 TO %RETRIES 'zerr=0 no error
9341|      REG %ax, &H1800
9342|      REG %cx, (( NoTracks - 1 ) * 256 ) + Info.ST
9343|      REG %dx, drive
9344|      CALL INTERRUPT &H13
9345|      zerr = REG( %ax ) \ 256 'zerr=&H0C unknown
    | media/maybe invalid CMOS
9346|      IF zerr THEN ResetFDC drive ELSE EXIT FOR
9347|      NEXT
9348|  'set INT 1Eh vector to this media's disk parameter
    | table in BIOS ROM
9349|    IF zerr = 0 THEN
9350|      NewDPTseg = REG( %es )
9351|      NewDPToff = REG( %di )
9352|      REG %ax, &H251E
9353|      REG %ds, NewDPTseg
9354|      REG %dx, NewDPToff
9355|      CALL INTERRUPT &H21
9356|    ELSE
9357|      OldDPTseg = 0
9358|      OldDPToff = 0
9359|    END IF
9360|  END IF
9361| ELSE
9362|  zerr = &HC 'media not supported by program
9363| END IF
9364| 'physical check for disk in the drive
9365| IF zerr = 0 THEN
9366|  REG %ax, &H401 'verify 1 sector from drive
9367|  REG %cx, &H1 'track=0 sector=1
9368|  REG %dx, drive 'head=0 drive=drive
9369|  FOR checks = 1 TO %RETRIES
9370|    CALL INTERRUPT &H13
9371|    IF REG(( %flags ) AND 1 ) THEN 'bad read
9372|      zerr = REG( %ax )
9373|  'need to detect an unformatted disk
9374|    SHIFT RIGHT zerr, 8 'e \ 256

```

```

9375|     ResetFDC drive
9376|     ELSE
9377|         zerr = 0 'good read,already formatted disk in
| drive
9378|     EXIT FOR
9379|     END IF
9380|     NEXT
9381| 'zerr may be any of the BIOS diskette error codes if
| non-zero here
9382| 'address mark not found(2)=unformatted disk in drive
9383| 'sector not found(4)=wacko disk but okay to proceed
9384|     IF zerr = 2 OR zerr = 4 THEN zerr = 0
9385|     END IF
9386|     ValidateDisk = zerr
9387| END FUNCTION
9388|
9389|
9390| 'format a track at a time a side at a time
9391| 'any error aborts format, diskette presumed unreliable
9392| 'retry format 1 or 2 more times before trashing the
| diskette
9393| FUNCTION FormatDisk(BYVAL drive AS INTEGER) AS INTEGER
9394|
9395|     DIM y AS INTEGER
9396|     DIM x AS INTEGER
9397|     DIM track AS INTEGER
9398|     DIM head AS INTEGER
9399|     DIM xerr AS INTEGER
9400|     DIM i AS INTEGER
9401|
9402|     y = POS( 0 )
9403|     x = CSRLIN
9404|     LOCATE ,,1
9405|     FOR track = 0 TO( NoTracks - 1 )
9406|         LOCATE x, y: PRINT "Formatting track: "; track;
9407|         'This is where you insert any printed info.
9408|         'If you don't want it, remove it.....
9409|         FOR head = 0 TO( Info.NH - 1 )
9410|             FOR i = 1 TO %RETRIES
9411|                 xerr = FormatTrack( drive, track, head )
9412|                 IF xerr = 0 THEN
9413|                     EXIT FOR
9414|                 END IF
9415|             NEXT
9416|             IF xerr THEN
9417|                 PRINT
9418|                 FUNCTION = xerr
9419|                 EXIT FUNCTION
9420|             END IF
9421|             NEXT

```

```

9422| NEXT
9423| PRINT
9424| FUNCTION = 0
9425| END FUNCTION
9426|
9427| FUNCTION WriteBoot(BYVAL drive AS INTEGER) AS INTEGER
9428|
9429|   DIM offset AS WORD
9430|   DIM i AS INTEGER
9431|   DIM Byte AS INTEGER
9432|   DIM xerr AS INTEGER
9433|
9434|   RESTORE BootSector
9435|
9436|   'read default boot record data
9437|   DEF SEG = VARSEG( BootRec ): offset = VARPTR( BootRec
    | )
9438|   FOR i = 0 TO 511
9439|     READ Byte
9440|     POKE offset + i, Byte
9441|   NEXT
9442|   DEF SEG
9443|
9444|   'update OEM name and BIOS parameter block
9445|   BootRec.parms = Info
9446|
9447|   'write the boot record
9448|   FOR i = 1 TO %RETRIES
9449|     xerr = WriteBootSector( drive )
9450|     IF xerr = 0 THEN EXIT FOR
9451|   NEXT
9452|
9453|   FUNCTION = xerr
9454|
9455| END FUNCTION
9456|
9457| FUNCTION WriteFAT(BYVAL drive AS INTEGER) AS INTEGER
9458|
9459|   DIM FAT1 AS STRING
9460|   DIM FAT2 AS STRING
9461|   DIM i AS INTEGER
9462|   DIM LogSec AS INTEGER
9463|   DIM j AS INTEGER
9464|   DIM k AS INTEGER
9465|   DIM xerr AS INTEGER
9466|   DIM cyl AS INTEGER
9467|   DIM hd AS INTEGER
9468|   DIM sec AS INTEGER
9469|
9470|   FAT1 = Info.MB + CHR$( 255 ) + CHR$( 255 )

```

```

9471| FAT2 = CHR$( 0 ) + CHR$( 0 ) + CHR$( 0 )
9472| FOR i = 1 TO 512: MID$( SectorBuff, i, 1 ) = CHR$( 0
| ): NEXT
9473| LogSec = Info.RS + Info.HS 'first logical FAT sector
9474| FOR i = 1 TO Info.NF
9475|   FOR j = 1 TO Info.SF
9476|     IF j = 1 THEN
9477|       MID$( SectorBuff, 1 ) = FAT1
9478|     ELSE
9479|       MID$( SectorBuff, 1 ) = FAT2
9480|     END IF
9481|     ComputeCHS LogSec, cyl, hd, sec
9482|     FOR k = 1 TO %RETRIES
9483|       xerr = WriteSector( drive, cyl, hd, sec,
| VARPTR32(SectorBuff) )
9484|       IF xerr = 0 THEN EXIT FOR
9485|     NEXT
9486|     IF xerr THEN
9487|       FUNCTION = xerr
9488|       EXIT FUNCTION
9489|     END IF
9490|     LogSec = LogSec + 1
9491|   NEXT
9492| NEXT
9493| FUNCTION = 0
9494| END FUNCTION
9495|
9496| FUNCTION WriteDir(BYVAL drive AS INTEGER) AS INTEGER
9497|
9498| DIM i AS INTEGER
9499| DIM LogSec AS INTEGER
9500| DIM k AS INTEGER
9501| DIM xerr AS INTEGER
9502| DIM cyl AS INTEGER
9503| DIM hd AS INTEGER
9504| DIM sec AS INTEGER
9505|
9506| MID$( SectorBuff, 1, 1 ) = CHR$( 0 )
9507| FOR i = 2 TO 512
9508|   IF ( i - 1 ) MOD 32 = 0 THEN
9509|     MID$( SectorBuff, i, 1 ) = CHR$( 0 )
9510|   ELSE
9511|     MID$( SectorBuff, i, 1 ) = CHR$( &HF6 )
9512|   END IF
9513| NEXT
9514| LogSec = Info.RS + Info.HS + ( Info.SF * Info.NF )
| 'first logical dir sector
9515| FOR i = 1 TO( Info.DE \ 16 ) 'sectors needed for root
| directory
9516|   ComputeCHS LogSec, cyl, hd, sec

```

```

9517|   FOR k = 1 TO %RETRIES
9518|     xerr = WriteSector( drive, cyl, hd, sec,
9519|       | VARPTR32(SectorBuff) )
9520|     IF xerr = 0 THEN EXIT FOR
9521|     NEXT
9522|     IF xerr THEN WriteDir = xerr: EXIT FUNCTION
9523|     LogSec = LogSec + 1
9524|     NEXT
9525|     WriteDir = 0
9526|   END FUNCTION
9527| 'set up media's format data
9528| SUB InitFormatParms(BYVAL media AS INTEGER)
9529|   Info.OEM = "IBM PB3" 'avoid changing 'IBM'
9530|   Info.BS = 512
9531|   Info.RS = 1
9532|   Info.NF = 2
9533|   Info.NH = 2
9534|   Info.HS = 0
9535|   SELECT CASE media
9536|     CASE %F360
9537|       Info.SC = 2
9538|       Info.DE = 112
9539|       Info.TS = 720
9540|       Info.MB = CHR$( %F360 )
9541|       Info.SF = 2
9542|       Info.ST = 9
9543|     CASE %F1200
9544|       Info.SC = 1
9545|       Info.DE = 224
9546|       Info.TS = 2400
9547|       Info.MB = CHR$( %F1200 )
9548|       Info.SF = 7
9549|       Info.ST = 15
9550|     CASE %F720
9551|       Info.SC = 2
9552|       Info.DE = 112
9553|       Info.TS = 1440
9554|       Info.MB = CHR$( ABS( %F720 ))
9555|       Info.SF = 3
9556|       Info.ST = 9
9557|     CASE %F1440
9558|       Info.SC = 1
9559|       Info.DE = 224
9560|       Info.TS = 2880
9561|       Info.MB = CHR$( %F1440 )
9562|       Info.SF = 9
9563|       Info.ST = 18
9564|     CASE %F2880
9565|       Info.SC = 2

```

```

9566|    Info.DE = 240
9567|    Info.TS = 5760
9568|    Info.MB = CHR$( ABS( %F2880 ))
9569|    Info.SF = 9
9570|    Info.ST = 36
9571| CASE ELSE
9572|    Info.OEM = ""
9573|    Info.BS = 0
9574|    Info.RS = 0
9575|    Info.NF = 0
9576|    Info.NH = 0
9577|    Info.HS = 0
9578|    Info.SC = 0
9579|    Info.DE = 0
9580|    Info.TS = 0
9581|    Info.MB = CHR$( 0 )
9582|    Info.SF = 0
9583|    Info.ST = 0
9584| END SELECT
9585| NoTracks = Info.TS \ Info.NH \ Info.ST
9586| NewDPTseg = 0    'INT 1Eh vector
9587| NewDPToff = 0    'new -> disk parameter table for
    | formatted media
9588| OldDPTseg = 0    'original-
9589| OldDPToff = 0    '-vector
9590| END SUB
9591|
9592| 'reset the controller after any BIOS FDC error
9593| SUB ResetFDC(BYVAL drive AS INTEGER)
9594| ! push DS
9595|
9596| ! mov AX, 0
9597| ! mov DX, drive
9598| ! int &H13
9599|
9600| ! pop DS
9601| END SUB
9602|
9603| FUNCTION FormatTrack(BYVAL drive AS INTEGER, _
9604|                     BYVAL track AS INTEGER, _
9605|                     BYVAL head AS INTEGER) AS INTEGER
9606|
9607| DIM sec AS INTEGER
9608| DIM cf AS INTEGER
9609| DIM e AS WORD
9610|
9611| 'Initialize address field for each sector on this track
9612| FOR sec = 1 TO Info.ST
9613|   AddrField( sec ).track = CHR$( track )
9614|   AddrField( sec ).head = CHR$( head )

```

```

9615|  AddrField( sec ).sector = CHR$( sec )
9616|  AddrField( sec ).bytesec = CHR$( 2 ) 'bytecode 2 =
    | 512-byte sector
9617|  NEXT
9618|
9619|  REG %ax, &H500 + Info.ST      'format track with
    | sectors/track
9620|  REG %cx,( track * 256 ) + 1    'track to
    | format,start with sector 1
9621|  REG %dx,( head * 256 ) + drive 'head,drive
9622|  REG %es, VARSEG( AddrField( 1 )) 'point to address
    | field data
9623|  REG %bx, VARPTR( AddrField( 1 ))
9624|  CALL INTERRUPT &H13
9625|
9626|  cf = REG( %flags ) AND 1      'cf=1 if disk error
9627|  IF cf THEN
9628|    E = REG( %ax )
9629|    SHIFT RIGHT e, 8            'return with status
    | byte
9630|    FUNCTION = e
9631|    ResetFDC drive
9632|  ELSE
9633|    IF verifydisk THEN
9634|      REG %ax, &H400 + Info.ST    'ok, verify track
    | integrity-
9635|      CALL INTERRUPT &H13        '-optional but
    | recommended on format
9636|      cf = REG( %flags ) AND 1    'cf=1 if disk
    | error
9637|      IF cf THEN
9638|        e = REG( %ax )
9639|        SHIFT RIGHT e, 8          'return with status
    | byte
9640|        FUNCTION = e
9641|        ResetFDC drive
9642|      END IF
9643|    END IF
9644|  END IF
9645|
9646|  FUNCTION = 0                  'format ok
9647|
9648| END FUNCTION
9649|
9650| 'convert a DOS logical sector to BIOS form
9651| SUB ComputeCHS( LogSec AS INTEGER, cyl AS INTEGER, hd
    | AS INTEGER, sec AS INTEGER)
9652|
9653| DIM CylSec AS INTEGER
9654| DIM rm AS INTEGER

```

```

9655|
9656|  CylSec = Info.ST * Info.NH
9657|  cyl   = LogSec \ CylSec
9658|  rm    = LogSec - ( cyl * CylSec )
9659|  hd    = rm \ Info.ST
9660|  sec   = rm - ( hd * Info.ST ) + 1
9661|
9662| END SUB
9663|
9664|
9665| FUNCTION WriteBootSector(BYVAL drive AS INTEGER) AS
    | INTEGER
9666|
9667|  DIM cf AS INTEGER
9668|  DIM e AS WORD
9669|
9670|  ! push DS          ; save DS for PowerBASIC
9671|
9672|  ! mov AX, &H301    ; write 1 sector
9673|  ! mov CX, 1        ; track 0, sector 1
9674|  ! mov DX, drive    ; head 0, drive
9675|  ! push SS
9676|  ! pop ES           ; ES:BX points to BootRec
9677|  ! lea BX, BootRec
9678|  ! int &H13         ; call BIOS
9679|  ! pop DS
9680|  ! jnc WriteBootDone
9681|  ! mov FUNCTION[0], AH ; AH holds error code
9682|
9683|  ResetFDC Drive    ' reset floppy controller
9684|
9685| WriteBootDone:
9686|
9687| END FUNCTION
9688|
9689| 'BIOS write a FAT or directory sector
9690| FUNCTION WriteSector( BYVAL drive AS INTEGER, _
9691|                      BYVAL cylinder AS INTEGER, _
9692|                      BYVAL head AS INTEGER, _
9693|                      BYVAL sector AS INTEGER, _
9694|                      BYVAL Buffer AS DWORD ) AS
    | INTEGER
9695|
9696|  ! push DS
9697|
9698|  ! mov AX, &H301
9699|  ! mov CL, Byte Ptr sector
9700|  ! mov CH, Byte Ptr cylinder
9701|  ! mov DL, Byte Ptr drive
9702|  ! mov DH, Byte Ptr head

```



```
9703| ! les BX, Buffer
9704| ! int &H13
9705| ! jnc WriteSectorDone
9706| ! xchg AL, AH
9707| ! xor AH, AH
9708| ! mov FUNCTION[0], AX
9709| ! push drive
9710| ! call WriteSector
9711| WriteSectorDone:
9712|
9713| ! pop DS
9714|
9715| END FUNCTION
9716|
9717| BootSector:
9718| DATA &HEB,&H3E,&H90,&H4D,&H53,&H57,&H49,&H4E,&H34,&H2E
9719| DATA &H30,&H00,&H02,&H01,&H01,&H00,&H02,&HE0,&H00,&H40
9720| DATA &H0B,&HF0,&H09,&H00,&H12,&H00,&H02,&H00,&H00,&H00
9721| DATA &H00,&H00,&H00,&H00,&H00,&H00,&H00,&H29,&HAA
9722| DATA &H16,&H9C,&HD4,&H4E,&H4F,&H20,&H4E,&H41,&H4D,&H45
9723| DATA &H20,&H20,&H20,&H20,&H46,&H41,&H54,&H31,&H32,&H20
9724| DATA &H20,&H20,&HF1,&H7D,&HFA,&H33,&HC9,&H8E,&HD1,&HBC
9725| DATA &HFC,&H7B,&H16,&H07,&HBD,&H78,&H00,&HC5,&H76,&H00
9726| DATA &H1E,&H56,&H16,&H55,&HBF,&H22,&H05,&H89,&H7E,&H00
9727| DATA &H89,&H4E,&H02,&HB1,&H0B,&HFC,&HF3,&HA4,&H06,&H1F
9728| DATA &HBD,&H00,&H7C,&HC6,&H45,&HFE,&H0F,&H8B,&H46,&H18
9729| DATA &H88,&H45,&HF9,&HFB,&H38,&H66,&H24,&H7C,&H04,&HCD
9730| DATA &H13,&H72,&H3C,&H8A,&H46,&H10,&H98,&HF7,&H66,&H16
9731| DATA &H03,&H46,&H1C,&H13,&H56,&H1E,&H03,&H46,&H0E,&H13
9732| DATA &HD1,&H50,&H52,&H89,&H46,&HFC,&H89,&H56,&HFE,&HB8
9733| DATA &H20,&H00,&H8B,&H76,&H11,&HF7,&HE6,&H8B,&H5E,&H0B
9734| DATA &H03,&HC3,&H48,&HF7,&HF3,&H01,&H46,&HFC,&H11,&H4E
9735| DATA &HFE,&H5A,&H58,&HBB,&H00,&H07,&H8B,&HFB,&HB1,&H01
9736| DATA &HE8,&H94,&H00,&H72,&H47,&H38,&H2D,&H74,&H19,&HB1
9737| DATA &H0B,&H56,&H8B,&H76,&H3E,&HF3,&HA6,&H5E,&H74,&H4A
9738| DATA &H4E,&H74,&H0B,&H03,&HF9,&H83,&HC7,&H15,&H3B,&HFB
9739| DATA &H72,&HE5,&HEB,&HD7,&H2B,&HC9,&HB8,&HD8,&H7D,&H87
9740| DATA &H46,&H3E,&H3C,&HD8,&H75,&H99,&HBE,&H80,&H7D,&HAC
9741| DATA &H98,&H03,&HF0,&HAC,&H84,&HC0,&H74,&H17,&H3C,&HFF
9742| DATA &H74,&H09,&HB4,&H0E,&HBB,&H07,&H00,&HCD,&H10,&HEB
9743| DATA &HEE,&HBE,&H83,&H7D,&HEB,&HE5,&HBE,&H81,&H7D,&HEB
9744| DATA &HE0,&H33,&HC0,&HCD,&H16,&H5E,&H1F,&H8F,&H04,&H8F
9745| DATA &H44,&H02,&HCD,&H19,&HBE,&H82,&H7D,&H8B,&H7D,&H0F
9746| DATA &H83,&HFF,&H02,&H72,&HC8,&H8B,&HC7,&H48,&H48,&H8A
9747| DATA &H4E,&H0D,&HF7,&HE1,&H03,&H46,&HFC,&H13,&H56,&HFE
9748| DATA &HBB,&H00,&H07,&H53,&HB1,&H04,&HE8,&H16,&H00,&H5B
9749| DATA &H72,&HC8,&H81,&H3F,&H4D,&H5A,&H75,&HA7,&H81,&HBF
9750| DATA &H00,&H02,&H42,&H4A,&H75,&H9F,&HEA,&H00,&H02,&H70
9751| DATA &H00,&H50,&H52,&H51,&H91,&H92,&H33,&HD2,&HF7,&H76
9752| DATA &H18,&H91,&HF7,&H76,&H18,&H42,&H87,&HCA,&HF7,&H76
```

```
9753| DATA &H1A,&H8A,&HF2,&H8A,&H56,&H24,&H8A,&HE8,&HD0,&HCC
9754| DATA &HD0,&HCC,&H0A,&HCC,&HB8,&H01,&H02,&HCD,&H13,&H59
9755| DATA &H5A,&H58,&H72,&H09,&H40,&H75,&H01,&H42,&H03,&H5E
9756| DATA &H0B,&HE2,&HCC,&HC3,&H03,&H18,&H01,&H27,&H0D,&H0A
9757| DATA &H49,&H6E,&H76,&H61,&H6C,&H69,&H64,&H20,&H73,&H79
9758| DATA &H73,&H74,&H65,&H6D,&H20,&H64,&H69,&H73,&H6B,&HFF
9759| DATA &H0D,&H0A,&H44,&H69,&H73,&H6B,&H20,&H49,&H2F,&H4F
9760| DATA &H20,&H65,&H72,&H72,&H6F,&H72,&HFF,&H0D,&H0A,&H52
9761| DATA &H65,&H70,&H6C,&H61,&H63,&H65,&H20,&H74,&H68,&H65
9762| DATA &H20,&H64,&H69,&H73,&H6B,&H2C,&H20,&H61,&H6E,&H64
9763| DATA &H20,&H74,&H68,&H65,&H6E,&H20,&H70,&H72,&H65,&H73
9764| DATA &H73,&H20,&H61,&H6E,&H79,&H20,&H6B,&H65,&H79,&H0D
9765| DATA &H0A,&H00,&H49,&H4F,&H20,&H20,&H20,&H20,&H20,&H20
9766| DATA &H53,&H59,&H53,&H4D,&H53,&H44,&H4F,&H53,&H20,&H20
9767| DATA &H20,&H53,&H59,&H53,&H80,&H01,&H00,&H57,&H49,&H4E
9768| DATA &H42,&H4F,&H4F,&H54,&H20,&H53,&H59,&H53,&H00,&H00
9769| DATA &H55,&HAA
9770|
9771|
9772|
9773| File Listing: NET_SETS.bat
9774|
9775| @echo Off
9776| rem *** Set default LAN connect parameters
9777| rem   this is mostly documentation for APP_SETS
9778| echo *** Setting Connection Parameters...
9779| echo *** Setting Connection Parameters...>>%LOGFILE%
9780| rem
9781| rem *** To use Static IP uncomment and change next two
    | lines
9782| rem SET SubNetMask=255 255 255 0
9783| rem SET IPAddress=192 168 1 222
9784| rem
9785| SET NETBIOSNAME=IPTEST
9786| rem SET NOLOGON=NOLOGON
9787| SET NETCARD=E100B
9788| SET SLOT=
9789| SET SERVER=ISSERVER
9790| SET USER=ISUSER
9791| SET PASSWORD=ISPASSWORD
9792| SET DOMAIN=ISDOMAIN
9793| SET WORKGROUP=ISWORKGROUP
9794| rem
9795| SET DRIVE1=
9796| SET SERVER1=
9797| SET SHARE1=
9798| SET DRIVE2=
9799| SET SERVER2=
9800| SET SHARE2=
9801| SET DRIVE3=
```

```
9802| SET SERVER3=
9803| SET SHARE3=
9804|
9805|
9806|
9807| File Listing: NETCARD.bat
9808|
9809| @echo %ToOff%
9810| rem check that we know how to do this card
9811|
9812| IF "%NETCARD%"=="3C90X" goto Found
9813| IF "%NETCARD%"=="3COM" goto Found
9814| IF "%NETCARD%"=="3C556" goto Found
9815| IF "%NETCARD%"=="NFLX3" goto Found
9816| IF "%NETCARD%"=="IBMFE" goto Found
9817| IF "%NETCARD%"=="E100B" goto Found
9818| IF "%NETCARD%"=="PCNTND" goto Found
9819| goto NoDriver
9820|
9821| :Found
9822| ECHO *** LAN CARD IS %NETCARD%
9823| ECHO *** LAN CARD IS %NETCARD% >>%LOGFILE%
9824|
9825| REM *** Create system.ini and protocol.ini
9826| copy %RAMD%\ini\s_base.ini + %RAMD%\ini\s_%NETCARD%.ini
    | %RAMD%\net\system.ini /a >%ToNul%
9827| copy %RAMD%\ini\p_%NETCARD%.ini %RAMD%\net\protocol.ini
    | >%ToNul%
9828|
9829| ECHO *** COMPUTER NAME IS %NETBIOSNAME% >>%LOGFILE%
9830| ECHO *** COMPUTER NAME IS %NETBIOSNAME%
9831|
9832| REM ***Change system.ini session settings
9833| gsr %RAMD%\NET\SYSTEM.INI "A:\NET" "%RAMD%\NET"
    | >%ToNul%
9834| gsr %RAMD%\NET\SYSTEM.INI "LOCALGROUP" "%WORKGROUP%"
    | >%ToNul%
9835| gsr %RAMD%\NET\SYSTEM.INI "DOMAINLOGON" "%USER%"
    | >%ToNul%
9836| gsr %RAMD%\NET\SYSTEM.INI "NETBIOSNAME" "%NETBIOSNAME%"
    | >%ToNul%
9837| IF "%IPAddress%"==" " GOTO Skip
9838| gsr %RAMD%\NET\PROTOCOL.INI "SubNetMask0="
    | "SubNetMask0=%SUBNETMASK%"
9839| gsr %RAMD%\NET\PROTOCOL.INI "IPAddress0="
    | "IPAddress0=%IPAddress%"
9840| gsr %RAMD%\NET\PROTOCOL.INI "DisableDHCP=0"
    | "DisableDHCP=1"
9841| :Skip
9842| IF "%SLOT%"==" " GOTO NoSlot
```

```
9843| gsr %RAMD%\NET\PROTOCOL.INI "SLOT=" "SLOT=%SLOT%"
9844| goto End
9845| :NoSlot
9846| gsr %RAMD%\NET\PROTOCOL.INI "SLOT=" ""
9847| goto END
9848|
9849| :NoDriver
9850| ECHO *** No driver for %NETCARD%. ***
9851| ECHO *** No driver for %NETCARD%. *** >>%LOGFILE%
9852| FATAL.BAT NO NIC DRIVER
9853|
9854| :NODETECT
9855| ECHO *** Failed to find a PCI NIC. ***
9856| ECHO *** Failed to find a PCI NIC >>%LOGFILE%
9857| FATAL.BAT NO NIC AUTODETECT
9858|
9859| :END
9860|
9861|
9862|
9863| File Listing: NETLOGON.bat
9864|
9865| @echo %ToOff%
9866| echo *** Logon to the Lan
9867| echo *** Logon to the Lan>>%LOGFILE%
9868| echo %PASSWORD% >%RAMD%\passwd.txt
9869| echo %PASSWORD% >>%RAMD%\passwd.txt
9870| net logon %USER% %PASSWORD% /Domain:%DOMAIN% /yes
    | /savepw:YES
9871| del %RAMD%\passwd.txt >%ToNul%
9872| :END
9873|
9874|
9875|
9876| File Listing: NETMAIN.bat
9877|
9878| @echo %ToOff%
9879| rem Call each of the sub modules
9880| if not "%NOLOGON%"==" " goto :end
9881| if "%result%" == "OK" call NETCARD.bat
9882| if "%result%" == "OK" call netstart.bat
9883| if "%result%" == "OK" call netlogon.bat
9884| if "%result%" == "OK" call connect.bat
9885| :end
9886|
9887|
9888|
9889| File Listing: NETSTART.bat
9890|
9891| @echo %ToOff%
```

```

9892| ECHO *** Starting LAN...
9893| echo *** Starting LAN...>>%LOGFILE%
9894| %RAMD%\net\net initialize >%ToNul%
9895| %RAMD%\net\net start NETBIND >%ToNul%
9896| %RAMD%\net\tcpsr.exe >%ToNul%
9897| %RAMD%\net\tinyrfc.exe <CRS.bat >%ToNul%
9898| :end
9899|
9900|
9901|
9902| File Listing: OK_WAIT.bat
9903|
9904| IF NOT "%OKWAIT%"=="OKWAIT" goto end
9905| :again
9906| echo .....DONE.....
9907| echo .
9908| echo . Remove the floppy and REBOOT now. .
9909| echo .
9910| echo .....
9911| fixboot /+
9912| echo .....DONE.....
9913| echo .
9914| echo . Remove the floppy and REBOOT now. .
9915| echo .
9916| echo .....
9917| fixboot /-
9918| echo .....DONE.....
9919| echo .
9920| echo . Remove the floppy and REBOOT now. .
9921| echo .
9922| echo .....
9923| fixboot /+
9924| echo .....DONE.....
9925| echo .
9926| echo . Remove the floppy and REBOOT now. .
9927| echo .
9928| echo .....
9929| fixboot /-
9930| goto again
9931| :end
9932|
9933|
9934|
9935| File Listing: P_e100b.ini
9936|
9937| [network.setup]
9938| version=0x3110
9939| netcard=$e100b,1,$E100B,1
9940| transport=tcpip,TCPIP
9941| lana0=$e100b,1,tcpip

```

9942|  
9943| [\$e100b]  
9944| DRIVENAME=E100B\$  
9945| SLOT=  
9946|  
9947| [protman]  
9948| drivename=PROTMAN\$  
9949|  
9950| [tcpip]  
9951| NBSessions=6  
9952| DefaultGateway0=  
9953| SubNetMask0=  
9954| IPAddress0=  
9955| DisableDHCP=0  
9956| DriverName=TCPIP\$  
9957| BINDINGS=\$e100b  
9958| LANABASE=0  
9959|  
9960|  
9961|  
9962| File Listing: RESULT.bat  
9963|  
9964| @echo %ToOff%  
9965| SET RESULT=OK  
9966|  
9967|  
9968|  
9969| File Listing: S\_e100b.ini  
9970|  
9971| [network drivers]  
9972| netcard=e100b.dos  
9973| transport=tcpdrv.dos,nemm.dos  
9974| devdir=A:\NET  
9975| LoadRMDrivers=yes  
9976|  
9977|  
9978|  
9979| File Listing: SETRAMD.bat  
9980|  
9981| @echo off  
9982| set RAMD=  
9983| a:\dos\findramd  
9984| if errorlevel 255 goto no\_ramdrive  
9985| if errorlevel 3 set RAMD=C:  
9986| if errorlevel 4 set RAMD=D:  
9987| if errorlevel 5 set RAMD=E:  
9988| if errorlevel 6 set RAMD=F:  
9989| if errorlevel 7 set RAMD=G:  
9990| if errorlevel 8 set RAMD=H:  
9991| if errorlevel 9 set RAMD=I:

```
9992| if errorlevel 10 set RAMD=J:
9993| if errorlevel 11 set RAMD=K:
9994| if errorlevel 12 set RAMD=L:
9995| if errorlevel 13 set RAMD=M:
9996| if errorlevel 14 set RAMD=N:
9997| if errorlevel 15 set RAMD=O:
9998| if errorlevel 16 set RAMD=P:
9999| if errorlevel 17 set RAMD=Q:
10000| if errorlevel 18 set RAMD=R:
10001| if errorlevel 19 set RAMD=S:
10002| if errorlevel 20 set RAMD=T:
10003| if errorlevel 21 set RAMD=U:
10004| if errorlevel 22 set RAMD=V:
10005| if errorlevel 23 set RAMD=W:
10006| if errorlevel 24 set RAMD=X:
10007| if errorlevel 25 set RAMD=Y:
10008| if errorlevel 26 set RAMD=Z:
10009| goto okay
10010|
10011| :no_ramdrive
10012| echo There has been an error booting your server. Boot
    | was not able
10013| echo to create a RAMDRIVE. To reboot,
10014| pause
10015| reboot
10016|
10017| :okay
10018|
10019|
10020|
10021| File Listing: STARTUP.bat
10022|
10023| @echo %ToOff%
10024| ECHO *** STARTUP.BAT ***
10025| @prompt $p$g
10026| set RESULT=OK
10027| rem call a:\dos\setramd.bat
10028| rem *** Setup path; APP is first so it may capture
    | calls
10029| path=%RAMD%\APP;%RAMD%\;%RAMD%\DOS;%RAMD%\NET;%RAMD%\INI
    | ;
10030| %ramd%
10031| rem ***
10032| rem *** Allow application to set LOGFILE, and signon
    | screen
10033| REM ***
10034| COPY A:\APP\*. * >%ToNul%
10035| rem LOGFILE = valid file name or NUL
10036| set LOGFILE=%RAMD%\LOG.TXT
10037| if exist %ramd%\APP_COPY.BAT call %ramd%\APP_COPY.BAT
```

```
10038| echo *** Copying files...
10039| ECHO *** Copying files... >>%LOGFILE%
10040| copy A:\*.bat >%ToNul%
10041| copy a:\*.cab %RAMD%\ >%ToNul%
10042| copy a:\fixboot.exe %RAMD%\ >%ToNul%
10043| rem
10044| echo *** Installing files...
10045| echo *** Installing files... >>%LOGFILE%
10046| rem
10047| md \dos
10048| cd \dos
10049| copy a:\dos\*. * >%ToNul%
10050| cd \
10051| extract /Y /E %RAMD%\data.cab >%ToNul%
10052| del %RAMD%\data.cab >%ToNul%
10053| ren %RAMD%\NET\LMHOSTS.SAM LMHOSTS >%ToNul%
10054| rem
10055| call Net_sets.bat
10056| if exist %ramd%\APP_SETS.BAT call %ramd%\APP_SETS.BAT
10057| call netmain.bat
10058| :end
10059|
10060|
10061|
10062| File Listing: TCPUTILS.ini
10063|
10064| [tcpglobal]
10065| drivename=GLOBAL$
10066|
10067| [sockets]
10068| drivename=SOCKETS$
10069| bindings=TCPIP_XIF
10070| numsockets=4
10071| numthreads=32
10072| poolsize=3200
10073| maxsendsize=1024
10074|
10075| [telnet]
10076| drivename=TELNET$
10077| bindings=TCPIP_XIF
10078| nsessions=0
10079| max_out_sends=0
10080|
10081|
10082|
10083| File Listing: WE_Wait.bat
10084|
10085| @if NOT "%WEWAIT%" == "WEWAIT" goto end
10086| :again
10087| fixboot /q
```



```

10088| if errorlevel 1 goto oops
10089| goto end
10090| :oops
10091| echo .....STOP.....
10092| echo . You must Write Enable the floppy to .
10093| echo .   update the status file.   .
10094| echo .....
10095| fixboot /-
10096| fixboot /s15
10097| fixboot /-
10098| fixboot /s15
10099| goto again
10100| :end
10101|
10102|
10103|
10104| File Listing: WP_Wait.bat
10105|
10106| @if NOT "%WPWAIT%" == "WPWAIT" goto end
10107| fixboot /q
10108| if errorlevel 4 goto waitloop
10109| if errorlevel 3 goto okay
10110| :waitloop
10111| echo .....STOP.....
10112| echo . You must Write Protect the floppy and .
10113| echo . reboot the computer to run restore. .
10114| echo .....
10115| fixboot /+
10116| fixboot /s5
10117| goto :waitloop
10118| :okay
10119| :end
10120|
10121|
10122|
10123| PSM Disaster Recovery Restore Source
10124|
10125| The Restore system is the DOS application that will
    | restore a previously backed up system drive. --LPW
10126|
10127|
10128|
10129| File Listing: CHECKSUM.pas
10130|
10131| unit CheckSum;
10132|
10133| interface
10134|
10135| const
10136|   CHECKSUM_IGNORE_DWORD = $f5e4d3c1;

```

```

10137|
10138|
10139| type
10140|   ChecksumEngine = object
10141|     (*----- data -----*)
10142|     SumAccumulator:   longint;
10143|     SumMultiplier:   longint;
10144|
10145|     (*----- methods -----*)
10146|     procedure Start (
10147|       DataSizeInBytes: word;
10148|       var Data: array of byte );
10149|
10150|     procedure Continue (
10151|       DataSizeInBytes: word;
10152|       var Data: array of byte );
10153|
10154|     function GetCheckSum: longint;
10155|   end;
10156|
10157|
10158| procedure WriteHex ( x: longint );
10159| procedure WriteHexByte ( x: byte );
10160|
10161| implementation
10162|
10163| const
10164|   CHECKSUM_MULTIPLIER = $1a2b3c4d;
10165|   CHECKSUM_INIT       = $c9d4b5a2;
10166|
10167| (*-----
   | -----*)
10168|
10169| procedure WriteHex ( x: longint );
10170| var i: integer;
10171|     n: integer;
10172| begin
10173|   for i := 0 to 7 do begin
10174|     n := (x SHR ((7-i)*4)) AND $0f;
10175|     if n < 10 then
10176|       write(n:1)
10177|     else
10178|       write(Chr(ord('a') + (n-10)));
10179|   end;
10180| end;
10181|
10182| (*-----
   | -----*)
10183|
10184| procedure WriteHexByte ( x: byte );

```

```

10185| var i: integer;
10186|   n: integer;
10187| begin
10188|   for i := 0 to 1 do begin
10189|     n := (x SHR ((1-i)*4)) AND $0f;
10190|     if n < 10 then
10191|       write(n:1)
10192|     else
10193|       write(Chr(ord('a') + (n-10)));
10194|   end;
10195| end;
10196|
10197|
10198| (*-----
   | -----*)
10199|
10200| procedure ChecksumEngine.Start (
10201|   DataSizeInBytes: word;
10202|   var Data:      array of byte );
10203| begin
10204|   SumAccumulator := CHECKSUM_INIT;
10205|   SumMultiplier  := CHECKSUM_MULTIPLIER;
10206|   Continue ( DataSizeInBytes, Data );
10207| end;
10208|
10209| (*-----
   | -----*)
10210|
10211| procedure ChecksumEngine.Continue (
10212|   DataSizeInBytes: word;
10213|   var Data:      array of byte );
10214| var
10215|   i: longint;
10216| begin
10217|   for i := 0 to longint(DataSizeInBytes)-1 do begin
10218|     SumAccumulator := SumAccumulator XOR ((($100 +
   | longint(Data[i])) * SumMultiplier));
10219|     INC (SumMultiplier);
10220|     SumAccumulator := (SumAccumulator SHL 9) OR
   | (SumAccumulator SHR (32-9));
10221|   end;
10222| end;
10223|
10224| (*-----
   | -----*)
10225|
10226| function ChecksumEngine.GetChecksum: longint;
10227| begin
10228|   if SumAccumulator = CHECKSUM_IGNORE_DWORD then
10229|     GetChecksum := $ffffff

```

```

10230| else
10231|     GetChecksum := SumAccumulator;
10232| end;
10233|
10234| begin {Unit initialization}
10235| end. {Unit initialization}
10236|
10237| (*--- end of file checksum.pas ---*)
10238|
10239|
10240|
10241| File Listing: DR.pas
10242|
10243| {$define NOLOGO}
10244| { $ define NOOVERLAY}
10245| {.define DEBUGOVERLAY}
10246| {$define OVERLAYSCSI}
10247| uses
10248|     {$ifndef NOOVERLAY}
10249|         overlay,
10250|         overinit,      { Overlay control }
10251|     {$endif}
10252|     dos,
10253|     versionu,
10254|     init,
10255|     vwinhigh,
10256|     vwinlow,
10257|     vtypes,
10258|     scsi,
10259|     scsihigh,
10260|     snaptypes,
10261|     vcrt,
10262|     bitedit,
10263|     vgen,
10264| { ini,  }
10265| { drivers,}
10266|     int13,
10267|     vbios,
10268|     VImage, CheckSum, Expand;
10269|
10270| {$ifndef NOOVERLAY}
10271|     {$ifndef DPML}
10272|         {$o viewfile}
10273|         {$o PCI}
10274|
10275|         {$o HugeNum}
10276|         {$o init}
10277|         {$o scsihigh}
10278|         {$o vdos}
10279|         {$o vdoshigh}

```

```

10280|     {$o vdates}
10281|     {$o vtext}
10282|     {$ifdef OVERLAYSCSI}
10283|     {$o vaspi}
10284|     {$o vcam}
10285|     {$o ncrscsii}
10286|     {$o Compaq}
10287|     {$o cpqpci}
10288|     {$o dell}
10289|     {$o mylex}
10290|     {$o mylex3}
10291|     {$o amipci}
10292|     {$endif}
10293| {$endif}
10294| {$endif}
10295|
10296|
10297| var
10298|   LogFile : Text;
10299| const
10300|   LogFileName = 'PSMDR.LOG';
10301|
10302| type
10303|   PartitionInfoPtr = ^VolumePartitionInformation;
10304| var
10305|   PartitionInfo: PartitionInfoPtr;
10306|
10307| type
10308|   optype = (manual,auto);
10309| var
10310|   opmode : optype;
10311|   Autotask : integer;
10312|
10313| type
10314|   int64parts = (lowpart,highpart);
10315|   int64 = array[int64parts] of longint;
10316|
10317|   pDeviceArray = ^tDeviceArray;
10318|   tDeviceArray = Array[1..10] of longint;
10319|
10320|   tVolume = record
10321|     Drive : pSCSIUnit;
10322|     VolStartSector : Longint;
10323| {   VolumeSizeinSectorsNTFS :   LongInt;  }
10324|     PartitionSizeinSectorsMBR :   LongInt;
10325| {   SerialNumber:   LongInt;  }
10326|   end;
10327|
10328|   Sourceltem = record
10329|     PathName : string;

```

```

10330|     State    : String;
10331|     StatusMsg : String;
10332|     VolSize   : int64;
10333| end;
10334| { tSourceImageList = Array[1..15] of SourceItem; }
10335| { pSourceImageList = ^tSourceImageList;      }
10336|
10337| const
10338|     MaxBackupDepth = 9;
10339|     MaxBackupLocations = 3;
10340|
10341| var
10342|     SourceImageList :
        | Array[1..MaxBackupLocations*MaxBackupDepth] of
        | SourceItem;
10343|     NumSourceImages : longint;
10344|     Target    : tVolume;
10345|     Task      : integer;
10346|
10347| const
10348|     ProgramLongName = 'PSM DR';
10349|
10350|     cValidate    =1;
10351|     cValidateAll =2;
10352|     cRecover     =3;
10353|     cFind        =4;
10354|
10355|     SECTOR_SIZE =512;
10356|
10357|     NumTasks = 4;
10358|     TaskList : Array[1..NumTasks] of string =
        | ('Validate','Validate All',
10359|         | 'Recover','Find');
10360|
10361|     TaskDone : Array[1..NumTasks] of string =
        | ('Valid','Valid',
10362|         | 'Restored','Exists');
10363|
10364|     TaskReached : Array[1..NumTasks] of string =
        | ('Validated','Validated',
10365|         | 'Restored','ThisisaBug');
10366|
10367|     TaskFail : Array[1..NumTasks] of string =
        | ('Failed','Failed',
10368|         | 'Failed','Failed');
10369|

```

```

10370| TaskDoneLog : Array[1..NumTasks] of string = (
10371|           'The backup image is
    | completely valid.',
10372|           'The backup image is
    | completely valid.',
10373|           'The backup image has been
    | restored and is rebootable.',
10374|           'The backup image has been
    | located.');
```

10375| { 'This message should not be  
| logged.'; }

```

10376|
10377|
10378| {
10379|     Callback procedure from init to display help when
    | the user does /? on the
10380|     command line
10381| }
10382| Procedure PSMDRHelp; far;
10383| Var
10384|     i : integer;
10385|     Ch : Char;
10386| Begin
10387|     i := 1;
10388|
10389|     while(i>0) do
10390|     Begin
10391|         clrscr;
10392|         writeln;
10393|         writeln(' PSM DR v1.0');
10394| {$ifdef NOCOPYRIGHT}
10395|         writeln('(c) Copyright 1989-2001 CDP, Inc.');
```

10396| {\$else}

```

10397|         writeln('(c) Copyright 1989-2001 Columbia Data
    | Products, Inc.');
```

10398| {\$endif}

```

10399|         writeln;
10400|
10401|         case i of
10402|             1: Begin
10403|                 writeln('Misc. Command line options');
10404|                 writeln;
10405|                 writeln('/?          = This screen');
10406|                 writeln;
10407|             end;
10408|             2: Begin
10409|                 writeln('SCSI and RAID Controller
    | command line options');
```

10410| writeln;

```

10411|                 writeln('/aspi = Force using ASPI');
```

```

10412|         writeln('/sdlp = Force using SDLP');
10413|         writeln('/compaq= Force using Compaq
| SMART Array');
10414|         writeln('/cpqpci= Force using Compaq
| PCI SMART Array 2');
10415|         writeln('/dell = Force using DELL
| RAID');
10416|         writeln('/mylex = Force using Mylex
| DAC960 Firmware version < 3.00');
10417|         writeln('/mylex3= Force using Mylex
| DAC960 Firmware version >= 3.00');
10418|         writeln('/amimega=Force using AMI
| MegaRAID');
10419|         writeln('/cam = Force using CAM');
10420|         writeln('/ncr = Force using NCR
| SCSI');
10421|         writeln('/int13 = Force using INT
| 13h');
10422|         writeln;
10423|         writeln('/wide = Scan target id"s
| 0-16 (Default is 0-7)');
10424|         writeln('/lun = Scan lun"s 0-7
| (Default is to scan LUN 0 only)');
10425|         writeln('/eb = Use alternate method
| of locating INT 13h devices');
10426|         writeln('/bios = Use the bios instead
| of cmos for drive types');
10427|         writeln;
10428|         end;
10429|     end;
10430|
10431|     writeln('Press <1> for general options, <2> for
| scsi options, <ESC> to quit');
10432|     ch := Readkey;
10433|     case ch of
10434|         '1' : i := 1;
10435|         '2' : i := 2;
10436|         #27 : i := 0;
10437|         #13,
10438|         '' : inc(i);
10439|     end;
10440|
10441|     if (i>2) then i:=1;
10442|
10443|     end;
10444|     Halt;
10445| End;
10446|
10447| Function DecideOpMode:BOOLEAN;
10448|

```



```

10449| var
10450|   ch      : Char;
10451|   DelayTicks : Longint;
10452|
10453| const
10454|   timelimit = 15;
10455|
10456| BEGIN
10457|
10458| { OpMode := auto;}
10459| if AutoBackup then
10460|   begin
10461|     WNewMsg('Splash',White,Blue,Blue,Cyan,
10462|       | ProgramLongName+'|'+
10463|         'Automatic
10464|         | processing will begin in xx seconds|'+
10465|         '|A keypress now
10466|         | or anytime during program|'+
10467|         'operation will
10468|         | return to manual control.|');
10469|
10470|     DelayTicks := BiosMemMap^.TimerTicksToday +
10471|       | (timelimit * 18);
10472|
10473|     Repeat
10474|       if keypressed then
10475|         Begin
10476|           { OpMode := manual;}
10477|           AutoBackup:=false;
10478|
10479|           Ch := ReadKey;
10480|           if upcase(ch)='P' then
10481|             Readkey
10482|             Else
10483|               Begin
10484|                 if ch = #0 then
10485|                   Readkey;
10486|                 DelayTicks := 0
10487|               End;
10488|             End;
10489|           gotoxy(38,03);
10490|           write((DelayTicks -
10491|             | BiosMemMap^.TimerTicksToday) div 18+1:2);
10492|           Until BiosMemMap^.TimerTicksToday >=

```

```

    | DelayTicks;
10493|
10494|     WDispose('Splash');
10495| end;
10496|
10497| { if Opmode=auto then}
10498| if AutoBackup then
10499| begin
10500|     { need code here to look at a:\ write protect
    | state
10501|     to decide if autovalidate or autorecover.
10502|     For now let's autorecover.
    | }
10503|     Task := cRecover;
10504| end;
10505|
10506| END;
10507|
10508| {
10509| I really need to move this to a library...
10510| }
10511| Function Zero ( W : Word ) : String;
10512| Var
10513| s : String;
10514| Begin
10515| Str(w:2,s);
10516| If s[1]=' ' Then
10517|     s[1] := '0';
10518| Zero := s;
10519| End;
10520|
10521| Function Hex ( x: longint ): string;
10522| var
10523| i: integer;
10524| n: integer;
10525| h: string;
10526|
10527| begin
10528| h := "";
10529| for i := 0 to 7 do begin
10530|     n := (x SHR ((7-i)*4)) AND $0f;
10531|     if n < 10 then
10532|         h := h + chr(ord('0')+n)
10533|     else
10534|         h := h + Chr(ord('a')+n-10);
10535|     end;
10536| Hex := h;
10537| end;
10538|
10539| Function HexByte ( x: longint ): string;

```

```

10540| var
10541|   i: integer;
10542|   n: integer;
10543|   h: string;
10544|
10545| begin
10546|   h := "";
10547|   for i := 0 to 1 do begin
10548|     n := (x SHR ((1-i)*4)) AND $0f;
10549|     if n < 10 then
10550|       h := h + chr(ord('0')+n)
10551|     else
10552|       h := h + Chr(ord('a')+n-10);
10553|   end;
10554|   HexByte := h;
10555| end;
10556|
10557|
10558|
10559| Procedure OpenLogFile;
10560| var
10561|   Open   : Boolean;
10562|
10563| BEGIN
10564|
10565|   System.assign (LogFile, LogFileName);
10566|   {$I-}
10567|   System.append (LogFile);
10568|   if System.IOresult <> 0 then begin
10569|     System.rewrite (LogFile);
10570|   end;
10571|   {$I+}
10572|
10573|
10574| {fixfixfix --- need to report to screen I guess...
10575| }
10576|
10577|   if System.IOresult <> 0 then begin
10578|     Open := false;
10579|   end else begin
10580|     Open := true;
10581|   end;
10582|
10583| end;
10584|
10585| Procedure CloseLogFile;
10586|
10587| BEGIN
10588|
10589|   {$I-}

```

```

10590|   System.close (LogFile);
10591|   {$!+}
10592|
10593| end;
10594|
10595| {
10596| }
10597| Function   VolBytesToSectors( NumBytes: int64 ) :
      | Longint;
10598|
10599| var
10600|   SectorUpper : longint;
10601|   SectorLower : longint;
10602|   SectorCarry : longint;
10603|
10604| Begin
10605|
10606| {
10607|   true 32+ bit voladdress handling code
10608|   NEED TO CHECK -1: not greater than 40 bit ( ie
      | sectornum < 32bit !!!!
10609| }
10610|   SectorUpper := (NumBytes[HighPart] shl 16 +
      | NumBytes[LowPart] shr 16 )
10611|
      | div SECTOR_SIZE;
10612|   SectorCarry := (NumBytes[HighPart] shl 16 +
      | NumBytes[LowPart] shr 16 )
10613|
      | mod SECTOR_SIZE;
10614|   SectorLower := (SectorCarry shl 16 +
      | NumBytes[LowPart] and $fff )
10615|
      | div SECTOR_SIZE;
10616| {
10617|   NEED TO CHECK -2:
10618|       0 = (SectorNumCarryPart1<<16 +
      | Prefix.UserData[0]&16 )
10619|
      | mod SECTOR_SIZE )
10620|   NEED TO CHECK -3:
10621|       0 = DataSize2<>0 and (DataSize1 mod
      | SECTOR_SIZE )<>0
10622| }
10623|   VolBytesToSectors := (SectorUpper shl 16) or
      | SectorLower;
10624|
10625| end;
10626|
10627|

```

```

10628| {$! fscommon.inc}
10629| {
10630|   Builds an MBR partition record for the target
      | volume.
10631| }
10632| Function   BuildPartitionRec( var Mbr: tMBR; var
      | VolSizeinBytes ) : Longint;
10633|
10634| var
10635|   VolSizeInSectors   : Longint;
10636|   VolLastSectorNTFS  : Longint;
10637|   PartitionLastSectorMBR : Longint;
10638|   Cylinder           : Word;
10639|
10640| Const
10641|   Partition1 : tPartRec = (
10642|       Bootable   : $80;
10643|       BeginHead  : $01;
10644|       BeginSector : $01;
10645|       BeginCyl   : $00;
10646|       FileSystem : cNTFS;
10647|       EndHead    : $99;
10648|       EndSector  : $aa;
10649|       EndCyl     : $bb;
10650|       StartSector : $cdefcdef;
10651|       SectorSize : $fdefcde );
10652|
10653| Begin
10654|
10655|   Mbr.Part[1] := Partition1;
10656|   Mbr.Part[1].StartSector := Target.Drive^.SPT;
10657|
10658|   VolSizeInSectors :=
      | VolBytesToSectors(int64(VolSizeinBytes));
10659|   { add Vol start which is start of 2nd absolute
      | track - i.e
10660|     allow for first track which the MBR has to itself
      | }
10661|   VolLastSectorNTFS := VolSizeInSectors +
      | Target.Drive^.SPT -1;
10662|
10663|   LBAToCHS( VolLastSectorNTFS,
10664|       Target.Drive^.SPT,
10665|       Target.Drive^.Heads,
10666|       Cylinder,
10667|       Mbr.Part[1].EndHead,
10668|       Mbr.Part[1].EndSector );
10669|
10670|   { looks like NOT NEEDED as Ntfs gives back the
      | true rounded partition }

```

```

10671|  {   leave for now as leads to setting value in
      | PartitionLastSectorMBR }
10672|
10673|  {Round Partition in MBR terms to a cylinder
      | boundary }
10674|  Mbr.Part[1].EndHead := Target.Drive^.Heads -1;
10675|  Mbr.Part[1].EndSector := Target.Drive^.SPT;
10676|
10677|  {Then need to reevaluate Last sector }
10678|  CHStoLBA( Cylinder,
10679|           Mbr.Part[1].EndHead,
10680|           Mbr.Part[1].EndSector,
10681|           Target.Drive^.SPT,
10682|           Target.Drive^.Heads,
10683|           PartitionLastSectorMBR );
10684|
10685|  Mbr.Part[1].SectorSize := PartitionLastSectorMBR
      | +1 - Target.Drive^.SPT;
10686|
10687|  ConvertCHSToPackedCHS(
10688|           Cylinder,
10689|           Mbr.Part[1].EndHead,
10690|           Mbr.Part[1].EndSector );
10691|
10692|  Mbr.Part[1].EndCyl := Cylinder;
10693|
10694|
10695| end;
10696|
10697| {
10698|   Find or create target Volume partition.
10699|   Handles validation and write if nec. of MBR.
10700|   Returns the partition's Start Sector number.
10701| }
10702| Function FindTargetVolume(var SourcePart :
      | VolumePartitionInformation; Action :longint ) :
      | Longint;
10703| var
10704|   VolSizeInSectors   : Longint;
10705|   VolStartSector     : Longint;
10706|   TargetMbr          : tMBR;
10707|   Err                : Longint;
10708|
10709| Const
10710|   DosMBR : Array[0..511] of byte = (
10711|     $FA,$33,$C0,$8E, $D0,$BC,$00,$7C,
      | $8B,$F4,$50,$07, $50,$1F,$FB,$FC,
10712|     $BF,$00,$06,$B9, $00,$01,$F2,$A5,
      | $EA,$1D,$06,$00, $00,$BE,$BE,$07,
10713|     $B3,$04,$80,$3C, $80,$74,$0E,$80,

```

| \$3C,\$00,\$75,\$1C, \$83,\$C6,\$10,\$FE,  
10714|     \$CB,\$75,\$EF,\$CD, \$18,\$8B,\$14,\$8B,  
| \$4C,\$02,\$8B,\$EE, \$83,\$C6,\$10,\$FE,  
10715|     \$CB,\$74,\$1A,\$80, \$3C,\$00,\$74,\$F4,  
| \$BE,\$8B,\$06,\$AC, \$3C,\$00,\$74,\$0B,  
10716|     \$56,\$BB,\$07,\$00, \$B4,\$0E,\$CD,\$10,  
| \$5E,\$EB,\$F0,\$EB, \$FE,\$BF,\$05,\$00,  
10717|     \$BB,\$00,\$7C,\$B8, \$01,\$02,\$57,\$CD,  
| \$13,\$5F,\$73,\$0C, \$33,\$C0,\$CD,\$13,  
10718|     \$4F,\$75,\$ED,\$BE, \$A3,\$06,\$EB,\$D3,  
| \$BE,\$C2,\$06,\$BF, \$FE,\$7D,\$81,\$3D,  
10719|     \$55,\$AA,\$75,\$C7, \$8B,\$F5,\$EA,\$00,  
| \$7C,\$00,\$00,\$49, \$6E,\$76,\$61,\$6C,  
10720|     \$69,\$64,\$20,\$70, \$61,\$72,\$74,\$69,  
| \$74,\$69,\$6F,\$6E, \$20,\$74,\$61,\$62,  
10721|     \$6C,\$65,\$00,\$45, \$72,\$72,\$6F,\$72,  
| \$20,\$6C,\$6F,\$61, \$64,\$69,\$6E,\$67,  
10722|     \$20,\$6F,\$70,\$65, \$72,\$61,\$74,\$69,  
| \$6E,\$67,\$20,\$73, \$79,\$73,\$74,\$65,  
10723|     \$6D,\$00,\$4D,\$69, \$73,\$73,\$69,\$6E,  
| \$67,\$20,\$6F,\$70, \$65,\$72,\$61,\$74,  
10724|     \$69,\$6E,\$67,\$20, \$73,\$79,\$73,\$74,  
| \$65,\$6D,\$00,\$00, \$00,\$00,\$00,\$00,  
10725|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10726|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10727|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10728|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10729|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10730|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10731|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10732|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10733|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10734|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10735|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10736|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10737|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
| \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,  
10738|     \$00,\$00,\$00,\$00, \$00,\$00,\$00,\$00,

```

    | $00,$00,$00,$00, $00,$00,$00,$00,
10739|    $00,$00,$00,$00, $00,$00,$00,$00,
    | $00,$00,$00,$00, $00,$00,$00,$00,
10740|    $00,$00,$00,$00, $00,$00,$00,$00,
    | $00,$00,$00,$00, $00,$00,$00,$00,
10741|    $00,$00,$00,$00, $00,$00,$00,$00,
    | $00,$00,$00,$00, $00,$00,$00,$00,
10742|    $00,$00,$00,$00, $00,$00,$00,$00,
    | $00,$00,$00,$00, $00,$00,$55,$AA
10743| );
10744|
10745|
10746| Begin
10747|
10748|   Target.PartitionSizeinSectorsMBR
10749|       :=
    | VolBytesToSectors(int64(SourcePart.PartitionLength));
10750|   Target.VolStartSector :=
    | VolBytesToSectors(int64(SourcePart.StartingOffset));
10751|
10752|   Err := Scsi_DasdRead( Target.Drive, 1, 0, 512,
    | @TargetMbr );
10753|   if Err=0 then
10754|   begin
10755|       {log what we found}
10756|       write (Logfile, 'MBR - SerialNum = 0x' );
10757|       write (Logfile, Hex(TargetMbr.SerialNumber));
10758|       writeln(Logfile);
10759|       write (Logfile, 'MBR - Partition ' );
10760|       write (Logfile,
    | HexByte(SourcePart.PartitionNumber));
10761|       write (Logfile, ': Type = 0x' );
10762|       write (Logfile, HexByte(TargetMbr.Part
10763|
    | [SourcePart.PartitionNumber].FileSystem));
10764|       write (Logfile, ' StartSector = 0x' );
10765|       write (Logfile, Hex(TargetMbr.Part
10766|
    | [SourcePart.PartitionNumber].StartSector));
10767|       write (Logfile, ' Size in sectors = 0x' );
10768|       write (Logfile, Hex(TargetMbr.Part
10769|
    | [SourcePart.PartitionNumber].SectorSize));
10770|       writeln(Logfile);
10771|       writeln(Logfile);
10772|
10773|       if ( TargetMbr.Id = cFDISKID ) and
10774|       (
    | TargetMbr.Part[SourcePart.PartitionNumber].FileSystem =
    | cNTFS ) and

```



```

10775|      (
      | TargetMbr.Part[SourcePart.PartitionNumber].SectorSize =
10776|      | Target.PartitionSizeInSectorsMBR ) and
10777|      (
      | TargetMbr.Part[SourcePart.PartitionNumber].StartSector
      | =
10778|      | Target.VolStartSector ) and
10779|      ( TargetMbr.SerialNumber =
      | SourcePart.DiskSerialNumber ) then
10780|
10781|      Begin    {use existing mbr}
10782| {      FindTargetVolume :=
10783|
      | TargetMbr.Part[SourcePart.PartitionNumber].StartSector;}
10784|      end else
10785|      begin
10786|          if (TargetMbr.Id <> cFDISKID) or
10787|
10788|          (( TargetMbr.Id = cFDISKID ) and
10789|          ( TargetMbr.Part[1].FileSystem = 0 )
      | and
10790|          ( TargetMbr.Part[2].FileSystem = 0 )
      | and
10791|          ( TargetMbr.Part[3].FileSystem = 0 )
      | and
10792|          ( TargetMbr.Part[4].FileSystem = 0 ))
      | then
10793|
10794|      Begin    {create new mbr with single
      | partition}
10795|          if Action = cRecover then
10796|          begin
10797|
      | move(DosMBR,TargetMbr,sizeof(tMBR));
10798|
      | BuildPartitionRec(TargetMbr,SourcePart.PartitionLength);
10799|          TargetMbr.SerialNumber :=
      | SourcePart.DiskSerialNumber;
10800|
10801|          Err := Scsi_DasdWrite(
      | Target.Drive, 1, 0, 512, @TargetMbr );
10802|          end;
10803|          end else
10804|          Begin    {Give up (for now). We daren't
      | overwrite the target}
10805|          Err := -1;
10806|          end;
10807|          end;

```

```

10808|   end;
10809|
10810|   FindTargetVolume := Err;
10811| End;
10812|
10813| var
10814|   GlobalScreenUpdate : Integer;
10815| {-----
| -----}
10816| {----- Code basically filched from VITEST.PAS
| -----}
10817| {----- (only amended to run as function
| instead of program) ---}
10818| {-----
| -----}
10819| Function ProcessChunk (
10820|   var Prefix: ChunkPrefix;
10821|   var Data1: array of byte;
10822|   var Data2: array of byte;
10823|   DataSize1: word;
10824|   DataSize2: word;
10825|   ImageNum : Longint;
10826|   Action: Longint ) : tSCSIError;
10827|
10828| var
10829|   Where:String;
10830|   Where2:String;
10831|   Err : tSCSIError;
10832|   SectorNumChunkPart1: Longint;
10833|   bigint : int64;
10834|   ch      : Char;
10835|
10836| begin
10837|
| {-----
| -----}
10838|   This procedure gets called every time a valid
| chunk is read.
10839|   Valid means that the correct amount of data was
| available in
10840|   the file, and that both the prefix and data
| checksums were
10841|   correct.
10842|
| -----
| -----}
10843|
10844|   Err := 0;
10845|
10846|   case Prefix.ChunkType of

```

```

10847|     VICT_START: begin
10848|         (* Put code here to process START chunk *)
10849|         writeln (Logfile, 'Found START chunk.' );
10850|
10851| {
10852|     SourceImageList[ImageNum].VolSize[LowPart]
10853|     | := Prefix.UserData[0];
10854|     SourceImageList[ImageNum].VolSize[HighPart]
10855|     | := Prefix.UserData[1];
10856| }
10857|     write (Logfile, 'Volume Size In Bytes   =
10858|         | 0x' );
10859|     Write (Logfile, Hex(Prefix.UserData[1]));
10860|     Write (Logfile, Hex(Prefix.UserData[0]));
10861|     writeln(Logfile);
10862|
10863|     write (Logfile, 'Number of Used Clusters =
10864|         | 0x' );
10865|     Write (Logfile, Hex(Prefix.UserData[3]));
10866|     Write (Logfile, Hex(Prefix.UserData[2]));
10867|     writeln(Logfile);
10868|
10869|     write (Logfile, 'Cluster Size in Bytes   =
10870|         | 0x' );
10871|     Write (Logfile, Hex(Prefix.UserData[4]));
10872|     writeln(Logfile);
10873|     writeln(Logfile);
10874|
10875| end;
10876|
10877|     VICT_PARTITION_INFO: begin
10878|         (* Put code here to process the volume's
10879|         | partition information *)
10880|
10881|         PartitionInfo := NIL;
10882|
10883|         writeln (Logfile, 'Found PARTITION_INFO
10884|         | chunk. (Data size is ',
10885|         |     DataSize1,
10886|         |     ', struct size is ',
10887|         |     sizeof(VolumePartitionInformation),
10888|         |     ')' );
10889|
10890|         PartitionInfo := PartitionInfoPtr(@Data1);
10891|
10892|         write (Logfile, 'Starting Offset   = 0x'
10893|         | );
10894|         write (Logfile,
10895|         |     Hex(PartitionInfo^.StartingOffset.HighPart));
10896|         write (Logfile,

```

```

    | Hex(PartitionInfo^.StartingOffset.LowPart));
10888|      writeln(Logfile);
10889|
10890|      write (Logfile, 'Partition Length    = 0x'
    | );
10891|      write (Logfile,
    | Hex(PartitionInfo^.PartitionLength.HighPart));
10892|      write (Logfile,
    | Hex(PartitionInfo^.PartitionLength.LowPart));
10893|      writeln(Logfile);
10894|
10895|      write (Logfile, 'Partition Number    = 0x'
    | );
10896|      write (Logfile,
    | Hex(PartitionInfo^.PartitionNumber));
10897|      writeln(Logfile);
10898|
10899|      write (Logfile, 'Partition Type      = 0x'
    | );
10900|      write (Logfile,
    | HexByte(PartitionInfo^.PartitionType));
10901|      writeln(Logfile);
10902|
10903|      write (Logfile, 'Boot Indicator      = 0x'
    | );
10904|      write (Logfile,
    | HexByte(PartitionInfo^.BootIndicator));
10905|      writeln(Logfile);
10906|
10907|      write (Logfile, 'Recognized Partition =
    | 0x');
10908|      write (Logfile,
    | HexByte(PartitionInfo^.RecognizedPartition));
10909|      writeln(Logfile);
10910|
10911|      write (Logfile, 'Reserved            =
    | 0x');
10912|      write (Logfile,
    | HexByte(PartitionInfo^.Reserved));
10913|      writeln(Logfile);
10914|
10915|      write (Logfile, 'DiskSerialNumber    = 0x');
10916|      write (Logfile,
    | Hex(PartitionInfo^.DiskSerialNumber));
10917|      writeln(Logfile);
10918|      writeln(Logfile);
10919|
10920|      SourceImageList[ImageNum].VolSize[HighPart]
    | :=
10921|

```

```

    | PartitionInfo^.PartitionLength.HighPart;
10922|      SourceImageList[ImageNum].VolSize[LowPart]
    | :=
10923|
    | PartitionInfo^.PartitionLength.LowPart;
10924|
10925| {      SourceImageList[ImageNum].SerialNumber :=
10926|
    | PartitionInfo^.DiskSerialNumber;}
10927|
10928|      Err := FindTargetVolume(PartitionInfo^,
    | Action);
10929|
10930|      end;
10931|
10932|      VICT_GRANULE: begin
10933|          (* Put code here to process GRANULE chunk
    | *)
10934|
10935| {      SectorNumChunkPart1 := Prefix.UserData[0]
    | div SECTOR_SIZE;}
10936|      BigInt[lowpart] := Prefix.UserData[0];
10937|      BigInt[highpart] := Prefix.UserData[1];
10938|      SectorNumChunkPart1 :=
    | VolBytesToSectors(BigInt);
10939|
10940|      if Action = cRecover then
10941|      begin
10942|          if DataSize1<>0 then
10943|          begin
10944|              Err := Scsi_DasdWrite(
    | Target.Drive,
10945|
    | (DataSize1+SECTOR_SIZE-1) div SECTOR_SIZE,
10946|
    | SectorNumChunkPart1 + Target.VolStartSector,
10947|              DataSize1,
10948|              @Data1
10949|          );
10950|
10951|          if (Err=0) and (DataSize2<>0) then
10952|          Begin
10953|              Err := Scsi_DasdWrite(
    | Target.Drive,
10954|
    | (DataSize2+SECTOR_SIZE-1) div SECTOR_SIZE,
10955|
    | SectorNumChunkPart1 + Target.VolStartSector
10956|
    | +(DataSize1 div SECTOR_SIZE) ,

```

```

10957|                DataSize2,
10958|                @Data2
10959|            );
10960|        End;
10961|
10962|        if Err<>0 then
10963|            Begin
10964|                write(Logfile, 'Error ',Err, '
| writing chunk starting at sector 0x');
10965|                Write(Logfile,
| Hex(SectorNumChunkPart1));
10966|                writeln(Logfile);
10967|            end;
10968|        end;
10969|    end;
10970|
10971|    inc(GlobalScreenUpdate);
10972|    if(GlobalScreenUpdate>=50) and (Err=0) then
10973|        Begin
10974|            GlobalScreenUpdate:=0;
10975|
10976|            str((SectorNumChunkPart1 div
| 2048),Where);
10977|            str(SectorNumChunkPart1,Where2);
10978|            Wmessage(TaskDone[Action] + ' up to '+
| Where + ' MB' + ' Sector='+Where2,White,Blue);
10979|            if keypressed then
10980|                begin
10981|                    while keypressed do Ch := ReadKey;
10982|                    writeln (LogFile, 'Operator pressed
| key at sector ', Where2);
10983|                    WNewMsg(' ',White,Blue,Blue,Cyan,
10984|                        ' The '+
| TaskList[Action] +
10985|                        ' has been
| interrupted from the keyboard.'|+
10986|                        '|Abandoning here
| will lose whatever has |+
10987|                        'been achieved so
| far and if recovering |+
10988|                        'may leave disk
| inconsistent!'+
10989|                        '||Press escape to
| continue the '+ TaskList[Action]+
10990|                        '|Any other key to
| abandon the '+ TaskList[Action]);
10991|
10992|                    Ch := ReadKey;
10993|                    if (ch<>chr(27)) then
10994|                        Err := -3;

```

```

10995|         while keypressed do Ch := ReadKey;
10996|
10997|         AutoBackup := false;
10998|         WDispose(' ');
10999|     end;
11000|     End;
11001| end;
11002|
11003|     VICT_FINISH: begin
11004|         (* Put code here to process FINISH chunk *)
11005|     end;
11006| end;
11007| ProcessChunk := Err;
11008| end;
11009|
11010| {
11011| procedure DumpPrefix ( var Prefix: ChunkPrefix );
11012| var i: integer;
11013| begin
11014|     writeln ( 'Prefix
| -----' );
11015|
11016|     write ( ' PrefixChecksum = ' );
11017|     WriteHex ( Prefix.PrefixChecksum );
11018|     writeln;
11019|
11020|     write ( ' ChunkChecksum = ' );
11021|     WriteHex ( Prefix.ChunkChecksum );
11022|     writeln;
11023|
11024|     write ( ' ChunkType = ' );
11025|     WriteHex ( Prefix.ChunkType );
11026|     writeln;
11027|
11028|     write ( ' PrefixSize = ' );
11029|     WriteHex ( Prefix.PrefixSizeInBytes );
11030|     writeln;
11031|
11032|     write ( ' ChunkSize = ' );
11033|     WriteHex ( Prefix.ChunkSizeInBytes );
11034|     writeln;
11035|
11036|     for i := 0 to VOLIMAGE_MAX_USER_DATA do begin
11037|         write ( ' UserData[', i:2, '] = ' );
11038|         WriteHex ( Prefix.UserData[i] );
11039|         writeln;
11040|     end;
11041|
11042|     for i := 0 to 6 do begin
11043|         write ( ' Reserved[', i:1, '] = ' );

```

```

11044|     WriteHex ( Prefix.Reserved[i] );
11045|     writeln;
11046| end;
11047|
11048|     writeln (
| '-----' );
11049| end;
11050| }
11051|
11052| function GetStatusMsg(status: tSCSIError): string;
11053| begin
11054|     case status of
11055|         -1:
| GetStatusMsg := 'Valid MBR - partition mismatch';
11056|         -2:
| GetStatusMsg := 'Source image not found';
11057|         -3:
| GetStatusMsg := 'Process cancelled by operator';
11058| {     CFE_READ_ERROR:
| GetStatusString := 'Error reading from input file';
11059|     CFE_OUT_OF_MEMORY:      GetStatusString
| := 'Out of memory';
11060|     CFE_CHECKSUM_FAILURE:   GetStatusString
| := 'Checksum failure';
11061|     CFE_UNKNOWN_COMPRESSION_TYPE: GetStatusString
| := 'Unknown data compression type';
11062|     CFE_CANNOT_OPEN_CONTINUATION: GetStatusString
| := 'Cannot open continuation file';
11063|     CFE_CHUNK_COUNTER_MISMATCH:  GetStatusString
| := 'Chunk counter mismatch';
11064|     CFE_MISSING_CONTINUATION_CHUNK: GetStatusString
| := 'Missing continuation chunk';
11065| }
11066|     else
| GetStatusMsg :=
| 'Unknown MyStatus';
11067| end;
11068| end;
11069|
11070| {
11071| }
11072| function ActOnImage(ImageNum : Longint ; Action
| :Longint ): boolean;
11073|
11074| type
11075|     ChunkDataPointer = ^ChunkDataBuffer;
11076|     ByteArray = array [0..0] of byte;
11077|
11078| var
11079|     Prefix:   ChunkPrefix;
11080|     Processor: ChunkFileProcessor;

```



```

11081|   FilePath: string;
11082|   Data1:   ChunkDataPointer;
11083|   Data2:   ChunkDataPointer;
11084|   DataSize1: word;
11085|   DataSize2: word;
11086|   Status:   ChunkFileStatus;
11087|   MyStatus: tSCSIError;
11088|   ChunkNumber: longint;
11089|   Happy, Finished: boolean;
11090|
11091|
11092| begin
11093|   new (Data1);
11094|   new (Data2);
11095|   Happy := true;
11096|   Finished := false;
11097|   Status := CFE_SUCCESS;
11098|
11099|   FilePath := SourceImageList[ImageNum].PathName;
11100|   ChunkNumber := 0;
11101|   if NOT Processor.OpenPath_SearchAllDrives(FilePath)
      | then begin
11102|       writeln (LogFile, 'Error: Could not open backup
      | set in path "', FilePath, '" ');
11103|       MyStatus := -2;
11104|       Happy := false;
11105|   end else begin
11106|       if action <> cFind then
11107|       begin
11108|           while Happy and not Finished do begin
11109|               Happy := Processor.GetNextChunk (
11110|                   Prefix,
11111|                   Data1^,
11112|                   Data2^,
11113|                   DataSize1,
11114|                   DataSize2,
11115|                   Status );
11116|
11117|               if not Happy then begin
11118|                   writeln (Logfile, '!!! Could not
      | read chunk number ', ChunkNumber );
11119|                   writeln (Logfile, '!!! Continuation
      | Number = ', Processor.ContinuationNumber );
11120|                   writeln (Logfile,
      | GetStatusString(Status) );
11121|               end else begin
11122|                   Happy :=
      | ExpandData(Prefix,Data1^,Data2^,DataSize1,DataSize2);
11123|                   if Happy then begin
11124|                       MyStatus := ProcessChunk (

```

```

    | Prefix, Data1^, Data2^, DataSize1, DataSize2, ImageNum,
    | Action );
11125|          if MyStatus <> 0 then begin
11126|              write (LogFile, 'Process
    | chunk Returned error ' );
11127|              write (Logfile,
    | Hex(MyStatus));
11128|              writeln(Logfile);
11129|              Happy := false;
11130|          end;
11131|              INC (ChunkNumber);
11132|          if Prefix.ChunkType =
    | VICT_FINISH then begin
11133|              writeln (LogFile, 'Found
    | FINISH chunk - backup image is complete.' );
11134|              Finished := true;
11135|          end;
11136|          end;
11137|          end;
11138|          end;
11139|          end;
11140|
11141|          Processor.Close;
11142|      end;
11143|
11144|      if Happy then begin
11145|          SourceImageList[ImageNum].State :=
    | TaskDone[Action];
11146|          writeln (LogFile, TaskDoneLog[Action] );
11147| {      if Action = cRecover then begin
11148|          SourceImageList[ImageNum].State :=
    | 'Restored';
11149|          writeln (LogFile, 'The backup image has
    | been restored and is rebootable.' );
11150|          end;
11151| }      end else begin
11152|          SourceImageList[ImageNum].State :=
    | TaskFail[Action];
11153|
11154| { FIXFIXFIX restore old status ??????? - can't if
    | doing restore!!!!!!
11155|      need to think this out - tomorrow!!!}
11156|      if Mystatus=-3 then
11157|          SourceImageList[ImageNum].State :=
    | TaskDone[cFind];
11158|
11159|      if finished then begin
11160|          writeln (LogFile, '!!! The backup image is
    | corrupt!');
11161| {          SourceImageList[ImageNum].State :=

```

```

    | 'Invalid';}
11162|     SourceImageList[ImageNum].StatusMsg :=
    | GetStatusString(Status);
11163|     end else begin
11164|         writeln (LogFile, '!!! The backup cannot be
    | used!' );
11165| {     SourceImageList[ImageNum].State :=
    | 'Invalid'; }
11166|     SourceImageList[ImageNum].StatusMsg :=
    | GetStatusMsg(Mystatus);
11167|     end;
11168| end;
11169|
11170| writeln (Logfile, 'Number of chunks processed   =
    | ', ChunkNumber );
11171| writeln (Logfile, 'Number of compressed granules =
    | ', Expand.GetNumCompressedGranules );
11172|
11173| dispose (Data1);
11174| dispose (Data2);
11175|
11176| ActOnImage := Happy;
11177| end;
11178|
11179| {-----
    | -----}
11180| {-----
    | -----}
11181| {----- END ....Code basically filched from
    | VITEST.PAS -----}
11182| {-----
    | -----}
11183| {-----
    | -----}
11184|
11185| {
11186| }
11187| Procedure MakeSourceImageList;
11188|
11189| var
11190|     i,j : Longint;
11191|     ImageName :string;
11192|     PathName   :string;
11193| const
11194|     HdgPSM = 'PSM Disaster Recovery';
11195|
11196| BEGIN
11197|
11198|     NumSourceImages:=0;
11199|     for i:=1 to MaxBackupDepth do

```

```

11200|   Begin
11201| {   ImageName := GetOption(
      | HdgPSM,'ImageName'+chr(i+48));}
11202| {   ImageName :=
      | GetEnv('ImageName'+chr(i+ord('0')));   }
11203| {   if ImageName <> " then
      | }
11204|   begin
11205|     for j:=1 to MaxBackupLocations do
11206|       begin
11207| {       PathName := GetOption(
          | HdgPSM,'PathName'+chr(j+48));}
11208|       PathName :=
          | GetEnv('Location'+chr(j+ord('0')));
11209|       if PathName <> " then
11210|         begin
11211|           NumSourceImages :=
              | NumSourceImages+1;
11212|           | SourceImageList[NumSourceImages].PathName :=
11213|           | PathName+'.'+chr(i+ord('0'));
11214|           ActOnImage(NumSourceImages,cFind);
11215|         end;
11216|       end;
11217|     end;
11218|   End;
11219|
11220|   if NumSourceImages=0 then
11221|     writeln (LogFile, 'No Source paths provided in
          | the environment variables.' );
11222|
11223| {
11224|   NumSourceImages := 5;
11225|
11226|   for i:=1 to 5 do
11227|     Begin
11228|       SourceImageList[i].PathName := '...\SourceFile'
          | +chr(i+48) + '.dri' ;
11229|       SourceImageList[i].PathName := 'vimage.dri';
11230|     End;
11231|
11232|     SourceImageList[4].PathName := 'vimage.dri';
11233|     SourceImageList[5].PathName := 'o:\vimage';
11234| }
11235|
11236| END;
11237|
11238| {
11239|   Finds which int13 device is the boot device.

```

```

11240| }
11241| Function GetTargetBootDevice( Count : Longint ) :
    | pSCSIUnit;
11242| var
11243|   i   : Longint;
11244|
11245| Begin
11246|
11247|   GetTargetBootDevice := nil;
11248|
11249|   for i:=1 to Count do
11250|     Begin
11251|
11252|       if ((Command.FakeDasd=0) and
11253|         (Dasd[i]^Method = cUseInt13) and
11254|         (Dasd[i]^Lun = 0) ) or
11255|
11256|         ((Command.FakeDasd>0) and
11257|         (Dasd[i]^Method = 0) and
11258|         (Dasd[i]^Lun = 1) )then
11259|
11260|         Begin
11261|           GetTargetBootDevice := Dasd[i];
11262|           break;
11263|         End;
11264|       End;
11265|
11266| End;
11267|
11268| {
11269|   Init the user interface, and scsi interface
11270| }
11271| Procedure Initialize;
11272|
11273| Var
11274|   S      : ST80;
11275|   Loopy  : INTEGER;
11276|   ch     : Char;
11277|   i      : Integer;
11278|   b      : Byte;
11279|   sunit  : pSCSIUnit;
11280|   Trans  : tTranslation;
11281|
11282| BEGIN
11283|
11284|   iVersion := IntToStr(BCDToDec(Hi(nVersion)))+'.'+
11285|     Zero(BCDToDec(Lo(nVersion)))+
11286|     Chr(nRevision+$60);
11287|
11288|   fillchar(Version,16,0);

```

```

11289| For i := 1 to Length(iVersion) do
11290|     Version[i-1]:=iVersion[i];
11291|
11292|
11293|
11294| {-----}
11295| { Here we accumulate all the information that }
11296| { will be passed on to the window init call. }
11297| {-----}
11298|
11299| Command.NoWarn    := False;
11300| Command.noNet     := False;
11301| Command.NewName    := "";
11302| Command.Graphics  := True;
11303| Command.Mono      := False;
11304| Command.SAllnitC   := 0;
11305| Command.ScanLuns   := False;
11306| Command.Bios       := False;
11307| Command.MaxTarget := 7;
11308|
11309|
11310| AutoBackUp := False;
11311| ReadRetry  := 2;
11312|
11313| S := "";
11314| DoSnapBackCommand := cDoNothing;
11315|
11316| { do default processing }
11317| ProcessCommandLine( Command, '@default.cfg',
    | PSMDRHelp );
11318|
11319| { now do command line processing }
11320| For Loopy := 1 to ParamCount Do
11321| Begin
11322|     ProcessCommandLine( Command, Paramstr( Loopy ),
    | PSMDRHelp );
11323| End; { For paramcount }
11324|
11325| S := S + 'SAVESCREEN,KHITS=2,LOOK=2,MOUSE,CLOCK,';
11326|
11327| { LoadIniFile('PSMDR.INI');}
11328|
11329| {-----}
11330| { Initialize the window library and the }
11331| { FILETEXT Unit.                      }
11332| {-----}
11333|
11334| If (LastMode = 7) or (CRTIsMono) then
11335| Begin
11336|     Command.Mono := True;

```

```

11337|   WinEnv.MenuLineColor :=0;
11338|   WinEnv.MenuLineColorF :=7;
11339| End;
11340|
11341| If Command.Mono Then
11342| Begin
11343|   CRTLoadMonoColorMap;
11344| End;
11345|
11346| If Not Command.Mono Then
11347| BEgin
11348|   If (command.graphics) and ((CrtIsVga) and
      | (ScreenRows=25)) Then
11349|     WOpen( ' ', Mgray, Mgray,
      | S+'PALETTE=BLUEGRAY,WIDGETFONT' )
11350|   Else
11351|     WOpen( '+-', black, blue, S);
11352|
11353|   WinEnv.Look := 2;
11354|
11355| End
11356| else
11357| BEgin
11358|   WOpen( '+-', LightGray, Black, S);
11359|   WinEnv.Border := 1;
11360|   WinEnv.Look := 0;
11361|   WinEnv.MenuLineColor :=0;
11362|   WinEnv.MenuLineColorF :=7;
11363| End;
11364|
11365| WCursorOFF;
11366|
11367| WPrgNameMsg(ProgramLongName+' Disaster Recovery '+'|3
      | Date 00/00/00 3 Time 00:00:00 AM',BLUE,CYAN);
11368|
11369| WinEnv.HelpFile:='diskedit.HLP';
11370| WSubmitHelpProc(@WNewHelp);
11371| WSubmitDefKeys;
11372|
11373| NumDev:=0;
11374|
11375| WMessage('Scanning for devices|3 Please
      | Wait...',WHITE,BLUE);
11376| WInfoMsg(' ',BLUE,CYAN);
11377|
11378| If Command.SAllInitC = 0 Then
11379| Begin
11380|   if Not GlobalWin95Installed Then
11381|     Command.SAllInitC := cUseAspiBit+
11382|                           cUseSdlpBit+

```

```

11383|          cUseCamBit+
11384|          cUseInt13Bit+
11385|          cUseNcrScsiBit+
11386|          cUseCompaqBit+
11387|          cUseCpqpciBit+
11388|          cUseDellBit+
11389|          cUseAmiMegaBit+
11390|          cUseMylex3Bit+
11391|          cUseMylexBit
11392|      Else
11393|          Command.SAllInitC := cUseAspiBit+
11394|          cUseSdlpBit+
11395|          cUseInt13Bit+
11396|          cUseNcrScsiBit+
11397|          cUseCompaqBit+
11398|          cUseCpqpciBit+
11399|          cUseDellBit+
11400|          cUseAmiMegaBit+
11401|          cUseMylex3Bit+
11402|          cUseMylexBit;
11403|  End;
11404|
11405|  IF (SCSI_Open( Command.SAllInitC )) Then
11406|      | SCSI_ScanSystem(Command.ScanLuns,Command.MaxTarget)
11407|  ELSE
11408|  BEGIN
11409|      writeln('Error no devices found');
11410|      Halt(5);
11411|  END;
11412|
11413|  if Command.FakeTape>0 then
11414|  Begin
11415|      for i := NumDev+1 to NumDev+Command.FakeTape+1 do
11416|      Begin
11417|          New(Sunit);
11418|          fillchar(Sunit^,sizeof(Sunit^),0);
11419|
11420|          Dev[i] := Sunit;
11421|          Sunit^.BlockSize := 32768;
11422|          Sunit^.VendorID := 'CDP  ';
11423|          Sunit^.ProdName := 'TEST TapeDrive ';
11424|          Sunit^.RevLevel := '1.0 ';
11425|          Sunit^.Name := Sunit^.VendorID +
            | Sunit^.ProdName + Sunit^.RevLevel;
11426|          Sunit^.DeviceType:= 1;
11427|          Sunit^.Host := 0;
11428|          Sunit^.Target := (i-1) div 8;
11429|          Sunit^.Lun := (i-1) mod 8;
11430|          Sunit^.Method := 0;

```



```

11431| End;
11432| NumDev := NumDev + Command.FakeTape;
11433| End;
11434|
11435| if Command.FakeDasd>0 then
11436| Begin
11437|   for i := NumDev+1 to NumDev+Command.FakeDasd+1 do
11438|     Begin
11439|       New(Sunit);
11440|       fillchar(Sunit^,sizeof(Sunit^),0);
11441|
11442|       Dev[i] := Sunit;
11443|       Sunit^.VendorID := 'CDP  ';
11444|       Sunit^.ProdName := 'TEST Hard Drive ';
11445|       Sunit^.RevLevel := '1.0 ';
11446|       Sunit^.Name := Sunit^.VendorID +
| Sunit^.ProdName + Sunit^.RevLevel;
11447|       Sunit^.DeviceType:= 0;
11448|       Sunit^.Host := 0;
11449|       Sunit^.Target := (i-1) div 8;
11450|       Sunit^.Lun := (i-1) mod 8;
11451|       Sunit^.Method := 0;
11452|       Sunit^.SPT := 63;
11453|       Sunit^.Heads := 255;
11454|       Sunit^.BlockSize := 512;
11455|       SUnit^.Blocks :=
| longint(Command.FakeDasdSize)*longint(2048);
11456|       SUnit^.DevSizeM := SUnit^.Blocks DIV
| (1048576 DIV SUnit^.BlockSize);
11457| End;
11458| NumDev := NumDev + Command.FakeDasd;
11459| End;
11460|
11461|
11462| WMessage('Making DASD device list|3 Please
| wait...',WHITE,BLUE);
11463|
11464| MakeDASDList;
11465| Target.Drive := GetTargetBootDevice(NumDASD);
11466|
11467| { Int13_GetTranslation(Target.Drive,@Trans);}
11468|
11469| {
11470| WNew( 3, 3,
11471|      60, 15,
11472|      BLUE,Cyan,
11473|      Cyan,Blue,
11474|      'Data' );
11475|
11476| writeln('Method=',Target.Drive^.Method,',

```

```

    | MType=',Target.Drive^.mtype,',
    | Host=',Target.Drive^.host);
11477|   writeln('target=',Target.Drive^.target,',
    | lun=',Target.Drive^.lun,',
    | channel=',Target.Drive^.channel);
11478|   writeln('Heads=',Target.Drive^.heads,',
    | spt=',Target.Drive^.spt,
11479|           ', blocks=',Target.Drive^.blocks,',
    | bs=',Target.Drive^.blocksize);
11480|   writeln('Vendorid=',Target.Drive^.vendorid,',
    | prodname=',Target.Drive^.prodname,',
    | revlevel=',Target.Drive^.revlevel);
11481|   writeln('name=',Target.Drive^.name);
11482|
11483|
11484|   WReadkey;
11485|   WDispose('Data');
11486| }
11487|   MakeSourceImageList;
11488|
11489|   WMessage(' ',White,Blue);
11490|   DecideOpMode;
11491|
11492| END;
11493|
11494| {-----
    | -----}
11495| {
11496|   Lets the user pick from a list of source images
11497| }
11498| Function SelectASourceImage( Header : String;
11499|                               x      : longint;
11500|                               y      : longint;
11501|                               Count  : Longint;
11502|                               DevicesToUse :
    | pDeviceArray ) : Longint;
11503| const
11504|   NumTokens = 4; {Num columns in menu table}
11505|
11506| var
11507|   m      : pMenuMax;
11508|   i      : integer;
11509|   Choice : Integer;
11510|   fs     : Word;
11511|   s      : String;
11512|   NumItems : Word;
11513|   LastHac : Byte;
11514|   RowTemplate : String;
11515|   TempStr : String;
11516|   ColumnSeq : Array [1..NumTokens] of Longint;

```

```

11517|
11518| Begin
11519|
11520|   Fs := 1;
11521|
11522|   New(m);
11523|   m^[1] := '~Source Volume Image Pathname   Status
      | Size Error Encountered   ';
11524|   m^[2] :=
      | '~AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
      | AAAAAAAAAAAAAAAAAA';
11525|   NumItems := 2;
11526|
11527|   RowTemplate := '%-30s%-8s%11d %-20s';
11528|
11529|   if Count>0 then
11530|   begin
11531|     for i:=1 to Count do
11532|     Begin
11533|
11534|       | str(SourcelmgeList[i].VolSize[LowPart]:11,TempStr);
11535|       m^[NumItems+1] :=
      | Pad(Copy(SourcelmgeList[i].PathName,1,30),30,OnRight,'
      | ') +
11536|       | Pad(Copy(SourcelmgeList[i].State,1,8),8,OnRight,' ') +
11537|       TempStr+' '+
11538|       | Pad(Copy(SourcelmgeList[i].StatusMsg,1,20),20,OnRight,'
      | ');
11539|
11540|
11541|   {
11542|     ColumnSeq[1] :=
      | Longint(@SourcelmgeList[i].PathName);
11543|     ColumnSeq[2] :=
      | Longint(@SourcelmgeList[i].State);
11544|     ColumnSeq[3] :=
      | SourcelmgeList[i].VolSize[LowPart];
11545|     ColumnSeq[4] :=
      | Longint(@SourcelmgeList[i].StatusMsg);
11546|     FormatStr(m^[NumItems+1], RowTemplate,
      | ColumnSeq);
11547|   }
11548|
11549|
11550|   {
11551|     for j:=1 to 40 do
11552|       S[j] := SourcelmgeList[i].PathName[j];

```

```

11553|
11554|
11555|
11556|      S := S +
      | ByteToHex(Dasd[DevicesToUse^[I]]^.Method) + ':' +
11557|      | ByteToHex(Dasd[DevicesToUse^[I]]^.MType) + ':' +
11558|      | IntToStr(Dasd[DevicesToUse^[I]]^.Host) + ':' +
11559|      | IntToStr(Dasd[DevicesToUse^[I]]^.Channel) + ':' +
11560|      | IntToStr(Dasd[DevicesToUse^[I]]^.Target) + ':' +
11561|      | IntToStr(Dasd[DevicesToUse^[I]]^.Lun);
11562|
11563|      S := S + ' -
      | '+Dasd[DevicesToUse^[I]]^.Name;
11564|    }
11565|      Inc(NumItems);
11566|    {   m^[NumItems] := s;}
11567|    End;
11568|
11569|    repeat
11570|      Choice := WAutoMenu (m,
11571|                          NumItems,
11572|                          1,
11573|
      | Lesserint(NumItems,WinEnv.LastRow-8),
11574|                          ",
11575|                          x,
11576|                          y,
11577|                          Blue,
11578|                          Cyan,
11579|                          Cyan,
11580|                          Blue,
11581|                          Header,
11582|                          fs);
11583|    until (choice>2) or (Choice<0);
11584|
11585|    Dispose(m);
11586|
11587|    if(Choice>0) then
11588|      SelectASourceImage := Choice-2
11589|    else
11590|      SelectASourceImage := -1;
11591|
11592|    end else
11593|    begin
11594|      WNewMsg(ProgramLongName,White,Blue,Blue,Cyan,

```

```

11595|                'No source image
      | paths have been defined. |+
11596|                'Define at least
      | one of the Environment Variables |+
11597|                '| "Location1"
      | "Location2" "Location3"||'+
11598|                'before calling
      | this program.|'+
11599|                '|||Press any key
      | to exit.|');
11600|
11601|    while not keypressed do ;
11602|
11603|        WDispose(ProgramLongName);
11604|        SelectASourceImage := -1;
11605|    End;
11606|
11607| End;
11608|
11609| {-----
      | -----}
11610| {
11611|    Lets the user pick task to run
11612| }
11613| Function SelectATask( Header : String;
11614|                    x    : longint;
11615|                    y    : longint;
11616|                    Count : Longint;
11617|                    DevicesToUse :
      | pDeviceArray ) : Longint;
11618| var
11619|    m    : pMenuMax;
11620|    i    : integer;
11621|    Choice : Integer;
11622|    fs    : Word;
11623|    s    : String;
11624|    NumItems : Word;
11625|    LastHac : Byte;
11626|
11627| Begin
11628|
11629|    Fs := 1;
11630|
11631|    New(m);
11632|    m^[1] := '~Action to perform  ';
11633|    m^[2] := '~AAAAAAAAAAAAAAAAAAAAA';
11634|    NumItems := 2;
11635|
11636|    for i:=1 to NumTasks do
11637|    Begin

```

```

11638|      S := TaskList[i];
11639| {
11640|      S := "";
11641|
11642|      S := S +
        | ByteToHex(Dasd[DevicesToUse^[i]]^.Method) + ':' +
11643|
        | ByteToHex(Dasd[DevicesToUse^[i]]^.MType) + ':' +
11644|
        | IntToStr(Dasd[DevicesToUse^[i]]^.Host) + ':' +
11645|
        | IntToStr(Dasd[DevicesToUse^[i]]^.Channel) + ':' +
11646|
        | IntToStr(Dasd[DevicesToUse^[i]]^.Target) + ':' +
11647|
        | IntToStr(Dasd[DevicesToUse^[i]]^.Lun);
11648|
11649|      S := S + ' - ' + Dasd[DevicesToUse^[i]]^.Name;
11650| }
11651|      Inc(NumItems);
11652|      m^[NumItems] := s;
11653| End;
11654|
11655|
11656| repeat
11657|      Choice := WAutoMenu (m,
11658|                          NumItems,
11659|                          1,
11660|
        | Lesserint(NumItems,WinEnv.LastRow-8),
11661|
        ",
11662|
        x,
11663|
        y,
11664|
        Blue,
11665|
        Cyan,
11666|
        Cyan,
11667|
        Blue,
11668|
        Header,
11669|
        fs);
11670| until (choice>2) or (Choice<0);
11671|
11672| Dispose(m);
11673|
11674|
11675|
11676| if(Choice>0) then
11677|      SelectATask := Choice-2
11678| else
11679|      SelectATask := 0;
11680| WDispose(Header);

```

```

11681| End;
11682|
11683| {-----
    | -----}
11684|
11685| procedure SuccessSound;
11686| var i: integer;
11687| const NumNotes = 3;
11688| begin
11689|     Delay(1); {calibrate timer}
11690|     for i := 1 to NumNotes do begin
11691|         Sound(440);
11692|         Delay(125);
11693|         Sound(880);
11694|         Delay(500);
11695|         NoSound;
11696|         if i < NumNotes then
11697|             Delay(1000);
11698|     end;
11699| end;
11700|
11701| {-----
    | -----}
11702|
11703| procedure FailureSound;
11704| var i: integer;
11705| const NumNotes = 1;
11706| begin
11707|     Delay(1); {calibrate timer}
11708|     for i := 1 to NumNotes do begin
11709|         Sound(220);
11710|         Delay(1000);
11711|         NoSound;
11712|         if i < NumNotes then
11713|             Delay(1000);
11714|     end;
11715| end;
11716|
11717| {-----
    | -----}
11718| procedure MainMenu;
11719| Var
11720|     Ch : Char;
11721|     i : Integer;
11722|     Sources : pDeviceArray;
11723|     SourceImage : integer;
11724| { Task : integer;}
11725|     Finished, Happy, TargetIntact: boolean;
11726|
11727| Begin

```

```

11728|   GetMem(Sources,NumDasd*sizeof(Longint));
11729|
11730|   SourcelImage := 0;
11731|
11732|   TargetIntact := true;
11733|   Happy := false;
11734|   Finished := false;
11735|   while not Finished do
11736|   Begin
11737| {
11738|     for i:= 1 to NumDASD do
11739|       Sources^[i] := i;
11740| }
11741|   GlobalScreenUpdate:=0;
11742|   WInfoMsg(' ',Blue,cyan);
11743|   Wmessage('Select a source image to process|3
| <ESC> Exits <F1> Help',White,Blue);
11744|
11745| {   if(OpMode = manual) then}
11746|   if not AutoBackup then
11747|   Begin
11748|     SourcelImage := SelectASourcelImage( 'Source
| Images', 1,3, NumSourcelImages, Sources );
11749|     if (SourcelImage<0) then
11750|     begin
11751|       if TargetIntact then
11752|       begin
11753|         Finished := true;
11754|       end else
11755|       begin
11756|         while keypressed do Ch := ReadKey;
11757|         WNewMsg(' ',White,Blue,Blue,Cyan,
11758|           ' An incomplete
| restore operation has left '+
11759|           ' the target drive
| compromised.'|'+
11760|           '| !! Exiting now
| may leave the device unbootable !! '|+
11761|           '||Press the "Y"
| key to exit anyway' +
11762|           '|Any other key
| will return to the main menu');
11763|
11764|         Ch := ReadKey;
11765|         if upcase(ch)='Y' then
11766|         begin
11767|           Finished := true;
11768|           writeln (LogFile, 'Operator
| confirmed to exit with drive compromised');
11769|         end;

```



```

11770|         while keypressed do Ch := ReadKey;
11771|
11772|         WDispose(' ');
11773|     end;
11774| end;
11775| End
11776| else
11777| Begin
11778|     SourcelImage := SourcelImage+1;
11779|     if(SourcelImage>NumSourcelImages) then
11780|         Finished := true;
11781| End;
11782|
11783| if (SourcelImage>0) and not Finished then
11784| Begin
11785|     Wmessage('Select an action to perform|3
| <ESC> Exits <F1> Help',White,Blue);
11786| {         if(OpMode = manual) then}
11787|         if not AutoBackup then
11788|             Begin
11789|                 Task := SelectATask( 'Action', 5,5
| ,NumDASD, Sources );
11790| {         End
11791|         else
11792|             Begin
11793|                 Task := AutoTask;
11794| }         End;
11795|         case Task of
11796|             cValidate  : begin
11797|                 Happy :=
| ActOnImage(SourcelImage,cValidate);
11798|             end;
11799|
11800|             cValidateAll : begin
11801|                 for i:= 1 to NumSourcelImages do
| begin
11802|                     ActOnImage(i,cValidate);
11803|                 end;
11804|             end;
11805|
11806|             cRecover    : begin
11807|                 TargetIntact := False;
11808|                 Happy :=
| ActOnImage(SourcelImage,cRecover);
11809|             if Happy then
11810|                 TargetIntact := True;
11811|             if AutoBackup then
11812|                 Finished := Happy;
11813|             end;
11814|

```

```

11815|         end;
11816|     End;
11817|
11818| End;
11819|
11820| if AutoBackup then begin
11821|     if Happy then begin
11822|         SuccessSound;
11823|     end else begin
11824|         FailureSound;
11825|     end;
11826| end;
11827|
11828| FreeMem( Sources,NumDasd*sizeof(Longint));
11829| End;
11830|
11831| {-----
    | -----}
11832|
11833| Procedure ShutDown;
11834| BEGIN
11835| {  UnLoadIniFile;}
11836|  WClose;
11837| END;
11838|
11839| Begin
11840|
11841|   OpenLogFile;
11842|
11843|   Initialize;
11844|
11845|   MainMenu;
11846|
11847|   ShutDown;
11848|
11849|   CloseLogFile;
11850|
11851| End.
11852|
11853|
11854|
11855| File Listing: expand.pas
11856|
11857| unit Expand;
11858|
11859| (*-----
    | -----*)
11860| interface
11861| uses VImage, CheckSum;
11862|

```

```

11863| function ExpandData (
11864|   var Prefix:   ChunkPrefix;
11865|   var Data1:    ChunkDataBuffer;
11866|   var Data2:    ChunkDataBuffer;
11867|   var DataSize1: word;
11868|   var DataSize2: word ): boolean;
11869|
11870| function GetNumCompressedGranules: longint;
11871|
11872| (*-----
   | -----*)
11873| implementation
11874|
11875| var NumCompressedGranules: longint;
11876|
11877| function GetNumCompressedGranules: longint;
11878| begin
11879|   GetNumCompressedGranules := NumCompressedGranules;
11880| end;
11881|
11882|
11883| function ExpandData (
11884|   var Prefix:   ChunkPrefix;
11885|   var Data1:    ChunkDataBuffer;
11886|   var Data2:    ChunkDataBuffer;
11887|   var DataSize1: word;
11888|   var DataSize2: word ): boolean;
11889| var
11890|   DataByte: byte;
11891|   i, ByteCount: longint; (* signed 32 bits because
   | we want 0-1 to be negative! *)
11892| begin
11893|   (*--- figure out what kind of data representation
   | is being used ---*)
11894|   if Prefix.ChunkType = VICT_GRANULE then begin
11895|     if Prefix.UserData[2] <> VI_COMPRESS_NONE then
   | begin
11896|       INC (NumCompressedGranules);
11897|     end;
11898|
11899|     case Prefix.UserData[2] of
11900|       VI_COMPRESS_NONE: begin
11901|         ExpandData := true; (* data is not
   | compressed *)
11902|       end;
11903|
11904|       VI_COMPRESS_ALL_BYTES_SAME: begin
11905|         if Prefix.UserData[4] <=
   | 2*MAX_CHUNK_BUFFER_SIZE then begin
11906|           ByteCount := Prefix.UserData[4];

```

```

11907|           DataByte :=
| byte(Prefix.UserData[3] AND $ff);
11908|           if ByteCount >
| MAX_CHUNK_BUFFER_SIZE then begin
11909|               (* need to split expanded data
| across 2 buffers *)
11910|               DataSize1 :=
| MAX_CHUNK_BUFFER_SIZE;
11911|               DataSize2 := ByteCount -
| MAX_CHUNK_BUFFER_SIZE;
11912|           end else begin
11913|               (* all the data will fit in the
| first buffer *)
11914|               DataSize1 := ByteCount;
11915|               DataSize2 := 0;
11916|           end;
11917|
11918|           for i := 0 to DataSize1-1 do begin
11919|               Data1[i] := DataByte;
11920|           end;
11921|
11922|           for i := 0 to DataSize2-1 do begin
11923|               Data2[i] := DataByte;
11924|           end;
11925|
11926|           ExpandData := true;
11927|       end else begin
11928|           (* data is too big! *)
11929|           ExpandData := false;
11930|       end;
11931|   end;
11932|
11933|   else begin
11934|       (* unknown compression algorithm! *)
11935|       ExpandData := false;
11936|   end;
11937| end;
11938| end else begin
11939|     ExpandData := true; (* no need to do anything;
| it's not a granule chunk *)
11940| end;
11941| end;
11942|
11943|
11944| begin {unit initialization}
11945|     NumCompressedGranules := 0;
11946| end. {unit initialization}
11947|
11948| (*--- end of file expand.pas ---*)
11949|

```

```

11950|
11951|
11952| File Listing: OVERINIT.pas
11953|
11954| unit Overinit;
11955|
11956| interface
11957|
11958| implementation
11959| uses
11960|     versionu,
11961|     overlay;
11962|
11963| Var
11964|     BufSize  : Longint;
11965|     FileName  : String;
11966|     Save      : Longint;
11967|     FSize     : Longint;
11968|     f         : File;
11969|     EmsRes    : Longint;
11970|
11971| { routines we need, as we cant call any overlaid unit.
    | }
11972| const
11973|     TDecHex : Array[0..15] of Char = '0123456789ABCDEF';
11974|
11975| Function ByteToHex(          B          : BYTE
    | ) : String;
11976| BEGIN
11977|     ByteToHex := TDecHex[(B AND $F0) SHR 4] + TDecHex[B
    | AND $0F];
11978| END;
11979|
11980| Function WordToHex(          W          : WORD
    | ) : String;
11981| BEGIN
11982|     WordTohex := ByteToHex( W SHR 8 ) + ByteToHex( W AND
    | $FF );
11983| END;
11984|
11985| Function StrToInt(          S          : STRING
    | ) : LONGINT;
11986| Var
11987|     Error : INTEGER;
11988|     Number : LONGINT;
11989| BEGIN
11990|     Val( S, Number, Error );
11991|     StrToInt := Number;
11992| END;
11993|

```

```

11994|
11995| Begin
11996|
11997|   OvrFileMode := 0 + 64; { Readonly-deny none }
11998|   BufSize := OvrGetBuf;
11999|
12000|   { find overlay file }
12001|   FileName:='dr.ovr';
12002|   ovrinit(filename);
12003|   if ovrResult<>0 then
12004|   Begin
12005|       writeln;
12006|       writeln('Overlay file not found. ');
12007|       halt(1);
12008|   End;
12009|
12010|   { for testing}
12011|
12012|   { Get File Size }
12013|
12014|   Save := FileMode;
12015|   FileMode := 0 + 64;
12016|   Assign(f,filename);
12017|   reset(f,1);
12018|   FSize := FileSize(f);
12019|   Close(f);
12020|   FileMode:=Save;
12021|
12022|   { init ems }
12023|   EmsRes :=ovrNoEMSDriver;
12024|
12025|   if(MemAvail>150000) then
12026|       BufSize := BufSize * 3
12027|   else
12028|       BufSize := BufSize * 2;
12029|
12030|   for Save:=1 to paramcount do
12031|   Begin
12032|       If pos('/overlay=', paramstr(Save) ) <> 0 Then
12033|       Begin
12034|           FileName :=
| copy(Paramstr(Save),pos('/overlay=',Paramstr(Save))+9,5)
| ;
12035|           BufSize := StrToInt(FileName);
12036|           if (BufSize<OvrGetBuf) then
12037|               BufSize := OvrGetBuf;
12038|       End;
12039|   End;
12040|
12041|   OvrSetBuf ( BufSize );

```

```

12042|   BufSize := OvrGetBuf;
12043|
12044|   case EmsRes of
12045|       ovrIOError   : writeln('       Overlay file
| I/O error.');
```

```

12046|       ovrNoEMSDriver: writeln('       EMS driver not
| installed. Using ',BufSize div 1024,'k of conventional
| memory.');
```

```

12047|       ovrNoEMSMemory: writeln('       Not enough EMS
| memory. Using ',BufSize div 1024,'k of conventional
| memory.');
```

```

12048|   else
12049|       writeln(FSize div 1024:7,'k of EMS memory
| allocated, ',BufSize div 1024,'k of conventional
| memory.');
```

```

12050|   end;
12051|
12052|
12053| End.
12054|
12055|
12056|
12057| File Listing: VIMAGE.pas
12058|
12059| unit VImage;
12060|
12061| interface
12062|
12063| uses Checksum;
12064|
12065| (*-----
| -----*)
12066|
12067| const
12068|   VOLIMAGE_MAX_USER_DATA = 20;
12069|   MAX_CHUNK_BUFFER_SIZE = WORD(32 * 1024);
12070|
12071|   {The following constants are Volume Image Chunk
| Types.}
12072|   {They will appears in Prefix.ChunkType.}
12073|
12074|   VICT_UNDEFINED      = 0; {should never be
| encountered in a file.}
12075|   VICT_START          = 1; {start of volume
| backup.}
12076|   VICT_GRANULE        = 2; {granule data.}
12077|   VICT_FINISH         = 3; {end of volume backup.}
12078|   VICT_PARTITION_INFO = 4; {partition information.}
12079|
12080|   VICT_RESERVED_BIT   = $8000;
```

```

12081|  VICT_BEGIN_CONTINUATION    = VICT_RESERVED_BIT OR
    | 1;
12082|  VICT_END_CONTINUATION      = VICT_RESERVED_BIT OR
    | 2;
12083|
12084|  {The following constants are data compression
    | algorithm types.}
12085|  {They appear only in granule chunks, in
    | UserData[2].}
12086|  VI_COMPRESS_UNDEFINED      = 0; {invalid
    | compression algorithm}
12087|  VI_COMPRESS_NONE           = 1; {the data is
    | not compressed - just plain}
12088|  VI_COMPRESS_ALL_BYTES_SAME = 2; {all the bytes
    | in the granule are the same value (UserData[3]=byte,
    | UserData[4]=count)}
12089|
12090| (*-----
    | -----*)
12091|
12092| type
12093|   BytePtr = ^byte;
12094|
12095|   ChunkPrefix = record
12096|     PrefixChecksum:   longint;
12097|     ChunkChecksum:    longint;
12098|     ChunkType:        longint;
12099|     PrefixSizeInBytes: longint;
12100|     ChunkSizeInBytes: longint;
12101|     UserData:         array
        | [0..VOLIMAGE_MAX_USER_DATA-1] of longint;
12102|     CumulativeChecksum: longint;
12103|     ChunkNumber:       longint;
12104|     Reserved:          array [0..4] of longint;
12105|   end;
12106|
12107|   ChunkDataBuffer = array
        | [0..MAX_CHUNK_BUFFER_SIZE-1] of byte;
12108|
12109|   ChunkFile = file;
12110|
12111|   ChunkFileStatus = (
12112|     CFE_SUCCESS,
12113|     CFE_READ_ERROR,
12114|     CFE_OUT_OF_MEMORY,
12115|     CFE_CHECKSUM_FAILURE,
12116|     CFE_UNKNOWN_COMPRESSION_TYPE,
12117|     CFE_CANNOT_OPEN_CONTINUATION,
12118|     CFE_CHUNK_COUNTER_MISMATCH,
12119|     CFE_MISSING_CONTINUATION_CHUNK

```



```

12120| );
12121|
12122| ChunkFileProcessor = object
12123|   (*----- data -----*)
12124|   InFile:          ChunkFile;
12125|   ContinuationNumber: integer;
12126|   IsOpen:          boolean;
12127|   BackupPath:      string;
12128|   CumulativeChecksum: longint;
12129|   ChunkCounter:    longint;
12130|   LastChunkType:   longint;
12131|   ChunkNumberInFile: longint;
12132|   IdentityString:  string;
12133|
12134|   (*----- methods -----*)
12135|   function OpenPath ( FilePath: string ):
      | boolean;
12136|   function OpenPath_SearchAllDrives ( FilePath:
      | string ): boolean;
12137|   procedure Close;
12138|
12139|   function GetNextChunk (
12140|       var Prefix:          ChunkPrefix;
12141|       var Data1:
      | ChunkDataBuffer;
12142|       var Data2:
      | ChunkDataBuffer;
12143|       var DataSize1:      word;
12144|       var DataSize2:      word;
12145|       var Status:         ChunkFileStatus
      | ): boolean;
12146|
12147|   function GetIdentityString: string;
12148| end;
12149|
12150| LargeInteger = record
12151|   LowPart: LongInt;
12152|   HighPart: LongInt;
12153| end;
12154|
12155| VolumePartitionInformation = record
12156|   StartingOffset:   LargeInteger;
12157|   PartitionLength:  LargeInteger;
12158|   PartitionNumber:  LongInt;
12159|   PartitionType:    Byte;
12160|   BootIndicator:    Byte; {nonzero if
      | volume is bootable}
12161|   RecognizedPartition: Byte; {nonzero if
      | partition is recognized}
12162|   Reserved:         Byte;

```

```

12163|     VolumeSerialNumber:   LongInt;
12164|     DiskSerialNumber:      LongInt;
12165| end;
12166|
12167| procedure SetChunkFileConsoleDebug ( NewDebugFlag:
    | boolean );
12168|
12169| function GetStatusString (status: ChunkFileStatus):
    | string;
12170|
12171| implementation
12172|
12173| var ConsoleDebug: boolean;
12174| procedure SetChunkFileConsoleDebug ( NewDebugFlag:
    | boolean );
12175| begin
12176|     ConsoleDebug := NewDebugFlag;
12177| end;
12178|
12179| (*-----
    | -----*)
12180|
12181| function GetStatusString (status: ChunkFileStatus):
    | string;
12182| begin
12183|     case status of
12184|         CFE_SUCCESS:           GetStatusString
    | := 'Success';
12185|         CFE_READ_ERROR:        GetStatusString
    | := 'Error reading from input file';
12186|         CFE_OUT_OF_MEMORY:      GetStatusString
    | := 'Out of memory';
12187|         CFE_CHECKSUM_FAILURE:   GetStatusString
    | := 'Checksum failure';
12188|         CFE_UNKNOWN_COMPRESSION_TYPE: GetStatusString
    | := 'Unknown data compression type';
12189|         CFE_CANNOT_OPEN_CONTINUATION: GetStatusString
    | := 'Cannot open continuation file';
12190|         CFE_CHUNK_COUNTER_MISMATCH: GetStatusString
    | := 'Chunk counter mismatch';
12191|         CFE_MISSING_CONTINUATION_CHUNK: GetStatusString
    | := 'Missing continuation chunk';
12192|         else                   GetStatusString
    | := 'Unknown error';
12193|     end;
12194| end;
12195|
12196| (*-----
    | -----*)
12197|

```

```

12198| function ContinuationString ( n: integer ): string;
12199| var
12200|   s: string;
12201|   i, denom: integer;
12202| begin
12203|   s := "";
12204|   denom := 100;
12205|   for i := 1 to 3 do begin
12206|     s := s + chr(ord('0') + ((n div denom) mod
12207|       | 10));
12207|     denom := denom div 10;
12208|   end;
12209|   ContinuationString := s;
12210| end;
12211|
12212| (*-----
12213|   | -----*)
12214| function GetChunkFileName ( BackupPath: string;
12215|   | ContinuationNumber: integer ): string;
12216| var FileName: string;
12217| begin
12218|   FileName := BackupPath;
12219|   if FileName = " then begin
12220|     FileName := '\';
12221|   end else if FileName[length(FileName)] <> '\' then
12222|   | begin
12223|     FileName := FileName + '\';
12224|   end;
12225|   FileName := FileName + 'image.' +
12226|   | ContinuationString(ContinuationNumber);
12227|   GetChunkFileName := FileName;
12228| end;
12229|
12227| (*-----
12228|   | -----*)
12229| function InternalOpen ( var cfp: ChunkFileProcessor ):
12230|   | boolean;
12231| var
12232|   SaveMode: byte;
12233|   FileName: string;
12234|   result: integer;
12235| begin
12236|   FileName := GetChunkFileName (cfp.BackupPath,
12237|   | cfp.ContinuationNumber);
12238|   if ConsoleDebug then begin
12239|     writeln ( '>>> InternalOpen: about to open "',
12240|     | FileName, '" ');
12241|   end;

```

```

12239|
12240|   SaveMode := System.FileMode;
12241|   System.FileMode := 0; {Allows us to open
    | read-only files with reset.}
12242|   System.assign (cfp.InFile, FileName);
12243|
12244|   result := System.IOresult;
12245|   if (result <> 0) and ConsoleDebug then begin
12246|       writeln ('>>> InternalOpen: Before open:
    | IOresult=', result);
12247|   end;
12248|
12249|   {$I-}
12250|   System.reset (cfp.InFile, 1);
12251|   {$I+}
12252|   result := System.IOresult;
12253|   if (result <> 0) and ConsoleDebug then begin
12254|       writeln ('>>> InternalOpen: System.IOresult
    | returned ', result );
12255|   end;
12256|   System.FileMode := SaveMode;
12257|   cfp.ChunkNumberInFile := 0;
12258|   InternalOpen := (result = 0);
12259| end;
12260|
12261| (*-----
    | -----*)
12262|
12263| function ParseSummaryLine (
12264|   Line:   string;
12265|   var Name: string;
12266|   var Value: string ): boolean;
12267| var
12268|   FoundEqual: boolean;
12269|   Valid: boolean;
12270|   i: integer;
12271| begin
12272|   FoundEqual := false;
12273|   Name := "";
12274|   Value := "";
12275|
12276|   for i := 1 to length(Line) do begin
12277|       if Line[i] = '=' then begin
12278|           FoundEqual := true;
12279|       end else begin
12280|           if FoundEqual then begin
12281|               Value := Value + Line[i];
12282|           end else begin
12283|               Name := Name + Line[i];
12284|           end;

```

```

12285|     end;
12286| end;
12287|
12288| Valid := FoundEqual and (Name <> "");
12289| if Valid and ConsoleDebug then begin
12290|     writeln ( '>>> ParseSummaryLine:
      | Name="",Name="", Value="",Value,"");
12291| end;
12292| ParseSummaryLine := Valid;
12293| end;
12294|
12295| (*-----
      | -----*)
12296|
12297| procedure ParseInteger ( s: string; var n: integer );
12298| var
12299|     ConvertCode: integer;
12300| begin
12301|     Val (s,n,ConvertCode);
12302|     if ConvertCode <> 0 then begin
12303|         n := 0;
12304|     end;
12305| end;
12306|
12307| (*-----
      | -----*)
12308|
12309| function DoSanityCheck ( FilePath: string; var
      | IdentityString: string ): boolean;
12310| var
12311|     SummaryFileName, BackupFileName: string;
12312|     SummaryFile: text;
12313|     BackupFile: file;
12314|     SaveMode: byte;
12315|     result: integer;
12316|     SummaryLine, SummaryName, SummaryValue: string;
12317|     NumBackupFiles, WarningCode, ImageNumber: integer;
12318| begin
12319|     (* Make sure we can open image.000, and that all
      | the image files it *)
12320|     (* describes indeed exist. *)
12321|
12322|     DoSanityCheck := false;
12323|     NumBackupFiles := 0;
12324|     WarningCode := 0;
12325|     IdentityString := "";
12326|     SummaryFileName := GetChunkFileName (FilePath, 0);
12327|     assign ( SummaryFile, SummaryFileName );
12328|     SaveMode := System.FileMode;
12329|     System.FileMode := 0;

```

```

12330|  {$I-}
12331|  System.reset (SummaryFile);
12332|  {$I+}
12333|  result := System.IOresult;
12334|  if ConsoleDebug then begin
12335|      writeln ( '>>> Summary File=',
12336|          | SummaryFileName,'" result=', result);
12337|  end;
12338|  if result = 0 then begin
12339|      while not EOF(SummaryFile) do begin
12340|          readln(SummaryFile,SummaryLine);
12341|          if
12342|              | ParseSummaryLine(SummaryLine,SummaryName,SummaryValue)
12343|              | then begin
12344|                  if SummaryName = 'NumFilesInBackup'
12345|                  | then begin
12346|                      ParseInteger (SummaryValue,
12347|                          | NumBackupFiles);
12348|                  end else if SummaryName = 'WarningCode'
12349|                  | then begin
12350|                      ParseInteger (SummaryValue,
12351|                          | WarningCode);
12352|                  end else if SummaryName = 'Identity'
12353|                  | then begin
12354|                      IdentityString :=
12355|                      IdentityString + SummaryLine + '
12356|                      '
12357|                  end
12358|              end
12359|          end
12360|      end
12361|  end
12362|  File Listing: VITEST.pas
12363|
12364| program TestVolumeImage;
12365|
12366| uses VImage, CheckSum, Expand;
12367|
12368| procedure ProcessChunk (
12369|     var Prefix: ChunkPrefix;
12370|     var Data1: array of byte;
12371|     var Data2: array of byte;
12372|     DataSize1: word;
12373|     DataSize2: word );
12374| type
12375|     PartitionInfoPtr = ^VolumePartitionInformation;
12376| var
12377|     PartitionInfo: PartitionInfoPtr;
12378| begin
12379|     {-----
12380|     -----

```

```

12370|     This procedure gets called every time a valid
      | chunk is read.
12371|     Valid means that the correct amount of data was
      | available in
12372|     the file, and that both the prefix and data
      | checksums were
12373|     correct.
12374|
      | -----
      | -----}
12375|     PartitionInfo := NIL;
12376|
12377|     case Prefix.ChunkType of
12378|     VICT_START: begin
12379|         (* Put code here to process START chunk *)
12380|         write ( 'Found START chunk. (' );
12381|         WriteHex (Prefix.ChunkType);
12382|         writeln('');
12383|
12384|         write ( 'Volume Size In Bytes  = 0x' );
12385|         WriteHex (Prefix.UserData[1]);
12386|         WriteHex (Prefix.UserData[0]);
12387|         writeln;
12388|
12389|         write ( 'Number of Used Clusters = 0x' );
12390|         WriteHex (Prefix.UserData[3]);
12391|         WriteHex (Prefix.UserData[2]);
12392|         writeln;
12393|
12394|         write ( 'Cluster Size in Bytes  = 0x' );
12395|         WriteHex ( Prefix.UserData[4] );
12396|         writeln;
12397|         writeln;
12398|     end;
12399|
12400|     VICT_PARTITION_INFO: begin
12401|         (* Put code here to process the volume's
      | partition information *)
12402|         writeln ( 'Found PARTITION_INFO chunk.
      | (Data size is ',
12403|             DataSize1,
12404|             ', struct size is ',
12405|             sizeof(VolumePartitionInformation),
12406|             ')' );
12407|
12408|         PartitionInfo := PartitionInfoPtr(@Data1);
12409|
12410|         write ( 'Starting Offset  = 0x' );
12411|         WriteHex (
      | PartitionInfo^.StartingOffset.HighPart );

```

```

12412|         WriteHex (
        | PartitionInfo^.StartingOffset.LowPart );
12413|         writeln;
12414|
12415|         write ( 'Partition Length   = 0x' );
12416|         WriteHex (
        | PartitionInfo^.PartitionLength.HighPart );
12417|         WriteHex (
        | PartitionInfo^.PartitionLength.LowPart );
12418|         writeln;
12419|
12420|         write ( 'Partition Number   = 0x' );
12421|         WriteHex ( PartitionInfo^.PartitionNumber
        | );
12422|         writeln;
12423|
12424|         write ( 'Partition Type     = 0x' );
12425|         WriteHexByte ( PartitionInfo^.PartitionType
        | );
12426|         writeln;
12427|
12428|         write ( 'Boot Indicator     = 0x' );
12429|         WriteHexByte ( PartitionInfo^.BootIndicator
        | );
12430|         writeln;
12431|
12432|         write ( 'Recognized Partition = 0x');
12433|         WriteHexByte (
        | PartitionInfo^.RecognizedPartition );
12434|         writeln;
12435|
12436|         write ( 'Reserved           = 0x');
12437|         WriteHexByte ( PartitionInfo^.Reserved );
12438|         writeln;
12439|
12440|         write ( 'VolumeSerialNumber = 0x');
12441|         WriteHex (
        | PartitionInfo^.VolumeSerialNumber );
12442|         writeln;
12443|
12444|         write ( 'DiskSerialNumber   = 0x');
12445|         WriteHex ( PartitionInfo^.DiskSerialNumber
        | );
12446|         writeln;
12447|
12448|         writeln;
12449|     end;
12450|
12451| VICT_GRANULE: begin
12452|     (* Put code here to process GRANULE chunk

```



```

    | *)
12453|     end;
12454|
12455|     VICT_FINISH: begin
12456|         (* Put code here to process FINISH chunk *)
12457|     end;
12458|
12459|     else begin
12460|         if Prefix.ChunkType =
            | VICT_BEGIN_CONTINUATION then begin
12461|             writeln ( 'Found BEGIN_CONTINUATION
            | chunk');
12462|         end else if Prefix.ChunkType =
            | VICT_END_CONTINUATION then begin
12463|             writeln ( 'Found END_CONTINUATION
            | chunk');
12464|         end else begin
12465|             write ( 'Found special chunk type: ');
12466|             WriteHex (Prefix.ChunkType);
12467|             writeln;
12468|         end;
12469|     end;
12470| end;
12471| end;
12472|
12473|
12474| procedure DumpPrefix ( var Prefix: ChunkPrefix );
12475| var i: integer;
12476| begin
12477|     writeln ( 'Prefix
            | -----' );
12478|
12479|     write ( ' PrefixChecksum = ');
12480|     WriteHex ( Prefix.PrefixChecksum );
12481|     writeln;
12482|
12483|     write ( ' ChunkChecksum = ');
12484|     WriteHex ( Prefix.ChunkChecksum );
12485|     writeln;
12486|
12487|     write ( ' ChunkType    = ');
12488|     WriteHex ( Prefix.ChunkType );
12489|     writeln;
12490|
12491|     write ( ' PrefixSize   = ');
12492|     WriteHex ( Prefix.PrefixSizeInBytes );
12493|     writeln;
12494|
12495|     write ( ' ChunkSize    = ');
12496|     WriteHex ( Prefix.ChunkSizeInBytes );

```

```

12497|   writeln;
12498|
12499|   for i := 0 to VOLIMAGE_MAX_USER_DATA do begin
12500|       write ( '   UserData[', i:2, '] = ' );
12501|       WriteHex ( Prefix.UserData[i] );
12502|       writeln;
12503|   end;
12504|
12505|   for i := 0 to 6 do begin
12506|       write ( '   Reserved[', i:1, '] = ' );
12507|       WriteHex ( Prefix.Reserved[i] );
12508|       writeln;
12509|   end;
12510|
12511|   writeln (
      | '-----' );
12512| end;
12513|
12514| type
12515|   ChunkDataPointer = ^ChunkDataBuffer;
12516|   ByteArray = array [0..0] of byte;
12517|
12518| var
12519|   Prefix:   ChunkPrefix;
12520|   Processor: ChunkFileProcessor;
12521|   FilePath: string;
12522|   Data1:    ChunkDataPointer;
12523|   Data2:    ChunkDataPointer;
12524|   DataSize1: word;
12525|   DataSize2: word;
12526|   Status:   ChunkFileStatus;
12527|   ChunkNumber: longint;
12528|   Happy, Finished: boolean;
12529|
12530| begin {program}
12531|   new (Data1);
12532|   new (Data2);
12533|   Happy := true;
12534|   Finished := false;
12535|   Status := CFE_SUCCESS;
12536|   VImage.SetChunkFileConsoleDebug (true);
12537|
12538|   if ParamCount = 0 then begin
12539|       writeln ( 'Use: vitest BackupFilePath' );
12540|   end else begin
12541|       FilePath := ParamStr(1);
12542|       if NOT
          | Processor.OpenPath_SearchAllDrives(FilePath) then begin
12543|           writeln ( 'Error: Could not open backup set
          | in path "', FilePath, '" );

```

```

12544|     end else begin
12545|         ChunkNumber := 0;
12546|         while Happy and not Finished do begin
12547|             Happy := Processor.GetNextChunk (
12548|                 Prefix,
12549|                 Data1^,
12550|                 Data2^,
12551|                 DataSize1,
12552|                 DataSize2,
12553|                 Status );
12554|
12555|             if not Happy then begin
12556|                 writeln ( '!!! Could not read chunk
| number ', ChunkNumber );
12557|                 writeln ( '!!! Continuation Number
| = ', Processor.ContinuationNumber );
12558|                 writeln ( GetStatusString(Status)
| );
12559|             end else begin
12560|                 Happy :=
| ExpandData(Prefix,Data1^,Data2^,DataSize1,DataSize2);
12561|                 if Happy then begin
12562|                     ProcessChunk ( Prefix, Data1^,
| Data2^, DataSize1, DataSize2 );
12563|                     INC (ChunkNumber);
12564|                     if Prefix.ChunkType =
| VICT_FINISH then begin
12565|                         writeln ( 'Found FINISH
| chunk - backup set is complete.' );
12566|                         Finished := true;
12567|                         end;
12568|                     end;
12569|                     end;
12570|                 end;
12571|
12572|                 Processor.Close;
12573|                 if Happy then begin
12574|                     writeln ( 'The backup set is completely
| valid.' );
12575|                 end else begin
12576|                     writeln ( '!!! The backup set is
| corrupt!' );
12577|                 end;
12578|
12579|                 writeln ( 'Number of chunks processed   =
| ', ChunkNumber );
12580|                 writeln ( 'Number of compressed granules =
| ', Expand.GetNumCompressedGranules )
12581|             end;
12582|         end;

```

```
12583|
12584|     dispose (Data1);
12585|     dispose (Data2);
12586|
12587| end. {program}
12588|
12589|
12590|
12591| PSM System DLL Source
12592|
12593| The DLL provides the user mode PSM services to the
    | system. --LPW
12594|
12595|
12596|
12597| File Listing: cacheloc.c
12598|
12599| #include <stdio.h>
12600| #include <stdlib.h>
12601| #include <string.h>
12602| #include <stddef.h>
12603| #include <windows.h>
12604| #include <tchar.h>
12605| #include <process.h>
12606| #include <time.h>
12607| #include <direct.h>
12608| #include <lm.h>
12609|
12610| #include <winioctl.h>
12611| #include <undoc.h>
12612| // PSM api
12613| #include <psm.h>
12614| // ioctls we need to send down
12615| #include "..\driver\ioctl.h"
12616|
12617| #include "volume.h"
12618|
12619| #include "defrag.h"
12620| #include <mountmgr.h>
12621| #include <ntddstor.h>
12622| #include <ntddvol.h>
12623|
12624| #include "setup5.h"
12625| #include "setup4.h"
12626| #include "service.h"
12627|
12628|
12629| // finds a volume that has enough disk space and
12630| // hopefully not being psmed.
12631|
```

```

12632| ULONG FindBestVolumeForCache (
12633|     pOpenTransactionIn3W In,
12634|     WCHAR *TempPath,
12635|     int TempPathSizeInChars )
12636| {
12637|     GetTempPathW(TempPathSizeInChars,TempPath);
12638|     return 0;
12639| }
12640|
12641|
12642|
12643| /*--- end of file cacheloc.c ---*/
12644|
12645|
12646|
12647| File Listing: CDP.h
12648|
12649| #define VER_COMPANYNAME_STR    "Columbia Data
    | Products, Inc."
12650| #define VER_LEGALTRADEMARKS_STR "PSM\256 is a
    | trademark of " VER_COMPANYNAME_STR
12651|
12652| #define VER_LEGALCOPYRIGHT_YEARS "1995-2001"
12653| #define VER_LEGALCOPYRIGHT_STR  "Copyright \251 "
    | VER_LEGALCOPYRIGHT_YEARS " " VER_COMPANYNAME_STR
12654|
12655| #if DBG
12656| #define VER_DEBUG              VS_FF_DEBUG
12657| #else
12658| #define VER_DEBUG              0
12659| #endif
12660|
12661| #if BETA
12662| #define VER_PRODUCTBETA_STR    "BETA"
12663| #define VER_PRERELEASE         VS_FF_PRERELEASE
12664| #else
12665| #define VER_PRERELEASE         0
12666| #define VER_PRODUCTBETA_STR    ""
12667| #endif
12668|
12669| #define VER_FILEFLAGSMASK      VS_FFI_FILEFLAGSMASK
12670| #define VER_FILEOS             VOS_NT_WINDOWS32
12671| #define VER_FILEFLAGS          (VER_PRERELEASE |
    | VER_DEBUG)
12672|
12673|
12674|
12675| File Listing: cluster.c
12676|
12677| #include <stdio.h>

```

```

12678| #include <stdlib.h>
12679| #include <string.h>
12680| #include <stddef.h>
12681| #include <windows.h>
12682| #include <tchar.h>
12683| #include <process.h>
12684| #include <time.h>
12685| #include <direct.h>
12686| #include <lm.h>
12687|
12688| #include <winioctl.h>
12689| #include <undoc.h>
12690| // psm api
12691| #include <psm.h>
12692| // ioctls we need to send down
12693| #include "..\driver\ioctl.h"
12694|
12695| #include "volume.h"
12696|
12697| #include "defrag.h"
12698| #include <mountdev.h>
12699| #include <ntddstor.h>
12700| #include <ntddvol.h>
12701| #include <aclapi.h>
12702| #include <clusapi.h>
12703|
12704| #include "cluster.h"
12705| #include <psm.h>
12706| #include <psmlapi.h>
12707| #include "dlog.h"
12708|
12709| #ifdef _DEBUG
12710|     #define STATIC
12711| #else
12712|     #define STATIC static
12713| #endif
12714|
12715| ULONG GetVolumeGuidForNtDeviceName( WCHAR *NTName,
    |   | WCHAR *VolumeGuid);
12716| PSMSTATUS PSMAPI Psmi_GetKernelSnapShotVolumesW (
12717|     PVOID KernelSnapShotPointer,
12718|     WCHAR *Buffer,
12719|     ULONG BufferSize );
12720|
12721|
12722| ULONG IsClusterInstalled(void)
12723| {
12724|     BOOL Ret=FALSE;
12725|     HMODULE Clusapi=LoadLibrary(TEXT("clusapi.dll"));
12726|     tGetNodeClusterState pGetNodeClusterState=NULL;

```

```

12727|
12728|  if((Clusapi!=INVALID_HANDLE_VALUE) &&
    | (Clusapi!=NULL)) {
12729|      pGetNodeClusterState =
    | (tGetNodeClusterState)GetProcAddress(Clusapi,"GetNodeClu
    | sterState");
12730|      if(pGetNodeClusterState) {
12731|          ULONG State=0;
12732|          pGetNodeClusterState(NULL,&State);
12733|          DLOG((TEXT("Cluster node state =
    | %08x\n"),State));
12734| /*
12735|      ClusterStateNotInstalled      The Cluster
    | service is not installed on the node.
12736|      ClusterStateNotConfigured    The Cluster
    | service is installed on the node but has not yet been
    | configured.
12737|      ClusterStateNotRunning        The Cluster
    | service is installed and configured on the node but is
    | not currently running.
12738|      ClusterStateRunning           The Cluster
    | service is installed, configured, and running on the
    | node.
12739| */
12740|      if((State == ClusterStateNotRunning) ||
    | (State==ClusterStateRunning)) {
12741|          DLOG((TEXT("Cluster node
    | configured\n")));
12742|          Ret = TRUE;
12743|      } else {
12744|          DLOG((TEXT("Cluster not
    | installed/configured\n")));
12745|      }
12746|      } else {
12747|          DLOG((TEXT("Error, entry point not found
    | for GetNodeClusterState\n")));
12748|      }
12749|      FreeLibrary(Clusapi);
12750|  } else {
12751|      DLOG((TEXT("Error %08x, unable to load
    | clusapi\n"),GetLastError()));
12752|  }
12753|  return Ret;
12754| }
12755|
12756|
12757| extern ULONG GetUniqueldForVolume (
12758|  HANDLE VolHandle,
12759|  WCHAR *Uniqueld,
12760|  ULONG UniqueldLength

```

```

12761| );
12762|
12763| ULONG ClusterGetApis( pClusApis Apis )
12764| {
12765|     ULONG Err=0;
12766|
12767|     memset(Apis,0,sizeof(tClusApis));
12768|
12769|     Apis->ClusApi=LoadLibrary(TEXT("clusapi.dll"));
12770|
12771|     if((Apis->ClusApi!=INVALID_HANDLE_VALUE) &&
        | (Apis->ClusApi!=NULL)) {
12772|         Apis->OpenCluster=
        | (tOpenCluster)GetProcAddress(Apis->ClusApi,"OpenCluster"
        | );
12773|         Apis->CloseCluster=
        | (tCloseCluster)GetProcAddress(Apis->ClusApi,"CloseCluste
        | r");
12774|         Apis->GetClusterKey=
        | (tGetClusterKey)GetProcAddress(Apis->ClusApi,"GetCluster
        | Key");
12775|         Apis->ClusterRegCreateKey =
        | (tClusterRegCreateKey)GetProcAddress(Apis->ClusApi,"Clus
        | terRegCreateKey");
12776|         Apis->ClusterRegOpenKey =
        | (tClusterRegOpenKey)GetProcAddress(Apis->ClusApi,"Cluste
        | rRegOpenKey");
12777|         Apis->ClusterRegDeleteKey =
        | (tClusterRegDeleteKey)GetProcAddress(Apis->ClusApi,"Clus
        | terRegDeleteKey");
12778|         Apis->ClusterRegCloseKey =
        | (tClusterRegCloseKey)GetProcAddress(Apis->ClusApi,"Clust
        | erRegCloseKey");
12779|         Apis->ClusterRegEnumKey =
        | (tClusterRegEnumKey)GetProcAddress(Apis->ClusApi,"Cluste
        | rRegEnumKey");
12780|         Apis->ClusterRegSetValue =
        | (tClusterRegSetValue)GetProcAddress(Apis->ClusApi,"Clust
        | erRegSetValue");
12781|         Apis->ClusterRegDeleteValue =
        | (tClusterRegDeleteValue)GetProcAddress(Apis->ClusApi,"Cl
        | usterRegDeleteValue");
12782|         Apis->ClusterRegQueryValue =
        | (tClusterRegQueryValue)GetProcAddress(Apis->ClusApi,"Clu
        | sterRegQueryValue");
12783|         Apis->ClusterRegEnumValue =
        | (tClusterRegEnumValue)GetProcAddress(Apis->ClusApi,"Clus
        | terRegEnumValue");
12784|         Apis->ClusterRegQueryInfoKey =
        | (tClusterRegQueryInfoKey)GetProcAddress(Apis->ClusApi,"C

```



```

    | lusterRegQueryInfoKey");
12785|     Apis->ClusterRegGetKeySecurity =
    | (tClusterRegGetKeySecurity)GetProcAddress(Apis->ClusApi,
    | "ClusterRegGetKeySecurity");
12786|     Apis->ClusterRegSetKeySecurity =
    | (tClusterRegSetKeySecurity)GetProcAddress(Apis->ClusApi,
    | "ClusterRegSetKeySecurity");
12787|     Apis->ClusterOpenEnum =
    | (tClusterOpenEnum)GetProcAddress(Apis->ClusApi,"ClusterO
    | penEnum");
12788|     Apis->ClusterEnum =
    | (tClusterEnum)GetProcAddress(Apis->ClusApi,"ClusterEnum"
    | );
12789|     Apis->ClusterCloseEnum =
    | (tClusterCloseEnum)GetProcAddress(Apis->ClusApi,"Cluster
    | CloseEnum");
12790|     Apis->GetNodeClusterState =
    | (tGetNodeClusterState)GetProcAddress(Apis->ClusApi,"GetN
    | odeClusterState");
12791|
12792|     Apis->GetClusterNodeState =
    | (tGetClusterNodeState)GetProcAddress(Apis->ClusApi,"GetC
    | lusterNodeState");
12793|     Apis->OpenClusterNode =
    | (tOpenClusterNode)GetProcAddress(Apis->ClusApi,"OpenClus
    | terNode");
12794|     Apis->CloseClusterNode =
    | (tCloseClusterNode)GetProcAddress(Apis->ClusApi,"CloseCl
    | usterNode");
12795|     Apis->ResumeClusterNode =
    | (tResumeClusterNode)GetProcAddress(Apis->ClusApi,"Resume
    | ClusterNode");
12796|     Apis->PauseClusterNode =
    | (tPauseClusterNode)GetProcAddress(Apis->ClusApi,"PauseCl
    | usterNode");
12797|
12798|     Apis->OpenClusterResource =
    | (tOpenClusterResource)GetProcAddress(Apis->ClusApi,"Open
    | ClusterResource");
12799|     Apis->CloseClusterResource =
    | (tCloseClusterResource)GetProcAddress(Apis->ClusApi,"Clo
    | seClusterResource");
12800|     Apis->OfflineClusterResource =
    | (tOfflineClusterResource)GetProcAddress(Apis->ClusApi,"O
    | fflineClusterResource");
12801|     Apis->OnlineClusterResource =
    | (tOnlineClusterResource)GetProcAddress(Apis->ClusApi,"On
    | lineClusterResource");
12802|     Apis->GetClusterResourceState =
    | (tGetClusterResourceState)GetProcAddress(Apis->ClusApi,"

```

```

    | GetClusterResourceState");
12803|     Apis->ClusterResourceControl =
    | (tClusterResourceControl)GetProcAddress(Apis->ClusApi,"C
    | lusterResourceControl");
12804|
12805| #ifdef DEBUG
12806|     // make sure everything loaded
12807|     {
12808|         ULONG i;
12809|         ULONG *Ptr;
12810|         Ptr = (ULONG*)Apis;
12811|         for(i=0;i<sizeof(tClusApis)/4;i++,Ptr++) {
12812|             if(!Ptr) {
12813|                 DLOG((TEXT("Error! Api %d is
    | missing!\n"),i));
12814|             }
12815|         }
12816|     }
12817| #endif
12818| } else {
12819|     Err = GetLastError();
12820|     DLOG((TEXT("Error %08x load module\n"),Err));
12821| }
12822| return Err;
12823| }
12824|
12825| ULONG ClusterCopyKey( pClusApis Apis, HKEY LocalKey,
    | HKEY ClusterKey )
12826| {
12827|     ULONG Err=0;
12828|     WCHAR Name[100];
12829|     ULONG NameLen;
12830|     ULONG Index=0;
12831|     ULONG Type;
12832|     ULONG DataSize;
12833|     PVOID Data;
12834|
12835|     do {
12836|         NameLen = sizeof(Name) / sizeof(WCHAR);
12837|         DataSize=0;
12838|         Err =
    | RegEnumValueW(LocalKey,Index,Name,&NameLen,NULL,NULL,NUL
    | L,&DataSize);
12839|         if(!Err) {
12840|             Data = LocalAlloc(LPTR,DataSize);
12841|             if(Data) {
12842|                 Err =
    | RegQueryValueExW(LocalKey,Name,NULL,&Type,Data,&DataSize
    | );
12843|                 if(!Err) {

```

```

12844|         Err =
|     Apis->ClusterRegSetValue(ClusterKey,Name,Type,Data,DataS
|     ize);
12845|         if(!Err) {
12846|             DLOG((TEXT("Success updating
|     %d:'%S', size=%d cluster\n"),Type,Name,DataSize));
12847|         } else {
12848|             DLOG((TEXT("Error %08x setting
|     data for value %S\n"),Err,Name));
12849|         }
12850|     } else {
12851|         DLOG((TEXT("Error %08x getting data
|     for value %S\n"),Err,Name));
12852|     }
12853|     LocalFree(Data);
12854| } else {
12855|     DLOG((TEXT("Error! out of memory for
|     data, need %d bytes\n"),DataSize));
12856|     Err = ERROR_OUTOFMEMORY;
12857| }
12858| } else {
12859|     if(Err!=ERROR_NO_MORE_ITEMS) {
12860|         DLOG((TEXT("Error %08x enumerating
|     key\n"),Err));
12861|     }
12862| }
12863|     Index++;
12864| } while(Err==0);
12865|
12866| if(Err==ERROR_NO_MORE_ITEMS) {
12867|     Err = 0;
12868| }
12869| return Err;
12870| }
12871|
12872| ULONG CopyKeyToClusterKey( WCHAR *VolumeGuid, WCHAR
|     *Uniqueld)
12873| {
12874|     tClusApis Apis;
12875|     HCLUSTER Cluster;
12876|     ULONG Err=0;
12877|     HKEY ClusterKey;
12878|     HKEY PSMKey;
12879|     HKEY psmlocalkey;
12880|     HKEY volumelocalkey;
12881|     ULONG Disp=0;
12882|
12883|     DLOG((TEXT("VolumeGuid='%S',
|     uniqueld='%S'\n"),VolumeGuid,Uniqueld));
12884|

```

```

12885| Err = ClusterGetApis(&Apis);
12886| if(!Err) {
12887|     Cluster = Apis.OpenCluster(NULL);
12888|     if(!Cluster) {
12889|         DLOG((TEXT("unable to open cluster\n")));
12890|         Err = GetLastError();
12891|     }
12892|     if(!Err) {
12893|         ClusterKey =
12894|         | Apis.GetClusterKey(Cluster,KEY_ALL_ACCESS);
12895|         if(!ClusterKey) {
12896|             DLOG((TEXT("unable to open root
12897|             | key\n")));
12898|             Err = GetLastError();
12899|         }
12900|         if(!Err) {
12901|             WCHAR Temp[200];
12902|             | wcsncpy(Temp,L"PersistentStorageManager\\");
12903|             wscat(Temp,Uniqueld);
12904|             __try {
12905|                 Err = Apis.ClusterRegCreateKey(
12906|                 | ClusterKey,
12907|                 | Temp,
12908|                 | REG_OPTION_NON_VOLATILE,
12909|                 | KEY_ALL_ACCESS,
12910|                 | NULL,
12911|                 | &PSMKey,
12912|                 | &Disp
12913|                 );
12914|             } __except (EXCEPTION_EXECUTE_HANDLER)
12915|             | {
12916|                 Err = GetExceptionCode();
12917|                 DLOG((TEXT("Exception %08x in
12918|                 | ClusterRegCreateKey\n"),Err));
12919|             }
12920|             if(!Err) {
12921|                 DLOG((TEXT("Disposition of PSM key
12922|                 | is %08x\n"),Disp));
12923|                 Err =
12924|                 | RegOpenKeyExW(HKEY_LOCAL_MACHINE,
12925|                 | L"SYSTEM\\CurrentControlSet\\Services\\PSMan5",
12926|                 | 0,
12927|                 | KEY_READ,
12928|                 | &psmlocalkey);
12929|                 if(!Err) {
12930|                     Err =

```

```

    | RegOpenKeyExW(psmlocalkey,
12927|         VolumeGuid,
12928|         0,
12929|         KEY_READ,
12930|         &volumelocalkey);
12931|         if(!Err) {
12932|             Err =
    | ClusterCopyKey(&Apis,volumelocalkey,PSMKey);
12933|
    | RegCloseKey(volumelocalkey);
12934|         } else {
12935|             DLOG((TEXT("Cluster: Error
    | %08x opening volume guid key\n"),Err));
12936|         }
12937|             RegCloseKey(psmlocalkey);
12938|         } else {
12939|             DLOG((TEXT("Cluster: Error %08x
    | opening psman5 key\n"),Err));
12940|         }
12941|             Apis.ClusterRegCloseKey(PSMKey);
12942|         } else {
12943|             DLOG((TEXT("Cluster: Error %08x
    | creating PSM key\n"),Err));
12944|         }
12945|             Apis.ClusterRegCloseKey(ClusterKey);
12946|         } else {
12947|             DLOG((TEXT("Cluster: Error %08x getting
    | cluster key\n"),Err));
12948|         }
12949|             Apis.CloseCluster(Cluster);
12950|         } else {
12951|             DLOG((TEXT("Cluster: Error %08x opening
    | cluster\n"),Err));
12952|         }
12953|             FreeLibrary(Apis.ClusApi);
12954|         } else {
12955|             DLOG((TEXT("Cluster: Error %08x unable to get
    | entry points\n"),Err));
12956|         }
12957|         return Err;
12958| }
12959|
12960| ULONG ClusterUpdateVolume( WCHAR *NTName )
12961| {
12962|     ULONG Err=0;
12963|     UNICODE_STRING Uni={0};
12964|     OBJECT_ATTRIBUTES ObjectAttributes={0};
12965|     HANDLE hVolume;
12966|     IO_STATUS_BLOCK IoStatus={0};
12967|     WCHAR VolumeGuid[120];

```

```

12968|  WCHAR Uniqueld[20*4];
12969|  ULONG UniqueldLength=sizeof(Uniqueld);
12970|
12971|  Err =
    | GetVolumeGuidForNtDeviceName(NTName,VolumeGuid);
12972|  if(Err) {
12973|      DLOG((TEXT("Error %08x volume guid\n"),Err));
12974|      return Err;
12975|  }
12976|
12977|  DLOG((TEXT("ClusterUpdateVolume: Nt  = '%S',
    | guid='%S'\n"),NTName,VolumeGuid));
12978|
12979|  RtlInitUnicodeString( &Uni, NTName);
12980|
12981|  InitializeObjectAttributes ( &ObjectAttributes,
12982|                              &Uni,
12983|                              OBJ_CASE_INSENSITIVE,
12984|                              NULL,
12985|                              NULL );
12986|
12987|  Err = NtCreateFile(  &hVolume,
12988|                     FILE_GENERIC_READ,
    | // desired access
12989|                     &ObjectAttributes,
    | // object attributes
12990|                     &IoStatus,
12991|                     NULL,          //
    | alloc size
12992|                     FILE_ATTRIBUTE_NORMAL,
    | // file attributes
12993|                     FILE_SHARE_WRITE |
    | FILE_SHARE_READ,          // share
    | access
12994|                     FILE_OPEN,      //
    | create disposition
12995|                     | FILE_SYNCHRONOUS_IO_NONALERT,          // create
    | options
12996|                     NULL, // eabuffer
12997|                     0 ); // ealength
12998|
12999|  if(!Err) {
13000|      Err =
    | GetUniqueldForVolume(hVolume,Uniqueld,sizeof(Uniqueld));
13001|      NtClose(hVolume);
13002|
13003|      if(!Err) {
13004|          // ClusterUpdateVolume: Nt  =
    | "\Device\HarddiskVolume1",

```

```

    | guid='Volume{0567ac62-a52a-11d4-a734-806d6172696f}'
13005|         VolumeGuid[43] = L'\0';
13006|         Err =
    | CopyKeyToClusterKey(VolumeGuid+7,UniqueId);
13007|     } else {
13008|         DLOG((TEXT("Error %08x getting unique
    | id\n"),Err));
13009|     }
13010| } else {
13011|     DLOG((TEXT("Error %08x opening
    | volume\n"),Err));
13012| }
13013| return Err;
13014| }
13015|
13016| ULONG UpdateClusterRegistries( tSnapShot *SnapShot )
13017| {
13018|     BOOL B=FALSE;
13019|     ULONG Err=0;
13020|     ULONG returned=0;
13021|
13022|     if(IsClusterInstalled()) {
13023|         if(Psm_IsPersistentSnapShotPointer(SnapShot)) {
13024|             WCHAR *Buffer;
13025|             ULONG BufferSize=128*1024;
13026|
13027|             Buffer = LocalAlloc(LPTR,BufferSize);
13028|             if(Buffer) {
13029|                 Err = Psmi_GetKernelSnapShotVolumesW(
13030|                     SnapShot,
13031|                     Buffer,
13032|                     BufferSize);
13033|
13034|                 if(!Err) {
13035|                     WCHAR *p=Buffer;
13036|
13037|                     while(*p) {
13038|                         Err = ClusterUpdateVolume(p);
13039|                         if(Err) {
13040|                             DLOG((TEXT("Error %08x
    | copying registry for volume '%S'\n"),p));
13041|                         }
13042|                         p+=wcslen(p)+1;
13043|                     }
13044|                 }
13045|                 LocalFree(Buffer);
13046|             } else {
13047|                 DLOG((TEXT("Out of memory for volume
    | list\n")));
13048|                 Err= ERROR_OUTOFMEMORY;

```

```

13049|     }
13050|   } else {
13051|     DLOG((TEXT("Not a valid snapshot
    | %08x\n"),SnapShot));
13052|     Err= ERROR_INVALID_PARAMETER;
13053|   }
13054| } else {
13055|   DLOG((TEXT("Cluster not installed\n")));
13056| }
13057|
13058| return Err;
13059| }
13060|
13061| ULONG ClusterGetNumberNodesActive()
13062| {
13063|   ULONG NumNodesActive=0;
13064|   ULONG NumNodes=0;
13065|   __try {
13066|     ULONG Err;
13067|     tClusApis Apis;
13068|     HCLUSTER Cluster;
13069|     HCLUSENUM Enum;
13070|
13071|     Err = ClusterGetApis(&Apis);
13072|     if(!Err) {
13073|       Cluster = Apis.OpenCluster(NULL);
13074|       if(!Cluster) {
13075|         DLOG((TEXT("unable to open
    | cluster\n")));
13076|         Err = GetLastError();
13077|       }
13078|       if(!Err) {
13079|
13080|         Enum =
    | Apis.ClusterOpenEnum(Cluster,CLUSTER_ENUM_NODE);
13081|
13082|         if(Enum) {
13083|           ULONG Index = 0;
13084|           ULONG Type;
13085|           ULONG Size;
13086|           WCHAR Buffer[64];
13087|
13088|           while(Err==0) {
13089|             Size = sizeof(Buffer) /
    | sizeof(WCHAR); // number of characters not bytes
13090|
13091|             Err = Apis.ClusterEnum(
13092|               Enum,
13093|               Index,
13094|               &Type,

```



```

13095|         Buffer,
13096|         &Size
13097|     );
13098|
13099|     if(Err==0) {
13100|         if(Type==CLUSTER_ENUM_NODE)
13101|         | {
13102|             ULONG State;
13103|             DLOG((TEXT("Node '%S'
13104| | found\n"),Buffer));
13105|
13106|             NumNodes++;
13107|
13108|             | Apis.GetNodeClusterState(Buffer,&State);
13109|             DLOG((TEXT(" Cluster
13110| | node state = %08x\n"),State));
13111|             /*
13112|             ClusterStateNotInstalled The
13113|             | Cluster service is not installed on the node.
13114|             ClusterStateNotConfigured The Cluster
13115|             | service is installed on the node but has not yet been
13116|             | configured.
13117|             ClusterStateNotRunning The
13118|             | Cluster service is installed and configured on the node
13119|             | but is not currently running.
13120|             ClusterStateRunning The
13121|             | Cluster service is installed, configured, and running
13122|             | on the node.
13123|             */
13124|             | if(State==ClusterStateRunning) {
13125|                 NumNodesActive++;
13126|             }
13127|             } else {
13128|                 DLOG((TEXT("'%'S' is not
13129| | a node\n"),Buffer));
13130|             }
13131|
13132|             Index++;
13133|         }
13134|     } // while(Err==0)
13135|
13136|     Apis.ClusterCloseEnum(Enum);
13137| } else {
13138|     DLOG((TEXT("Cluster: Error opening
13139| | enum for nodes\n")));
13140| }
13141|
13142| Apis.CloseCluster(Cluster);

```

```

13131|         } else {
13132|             DLOG((TEXT("Cluster: Error %08x opening
| cluster\n"),Err));
13133|         }
13134|         FreeLibrary(Apis.ClusApi);
13135|     } else {
13136|         DLOG((TEXT("Cluster: Error %08x unable to
| get entry points\n"),Err));
13137|     }
13138| } __except(EXCEPTION_EXECUTE_HANDLER) {
13139|     DLOG((TEXT("Exception %08x
| ClusterGetNumberNodesActive\n"),GetExceptionCode()));
13140| }
13141| DLOG((TEXT("Cluster: Found %d nodes, of which %d
| are active\n"),NumNodes,NumNodesActive));
13142| return NumNodesActive;
13143| }
13144|
13145|
13146| /*
13147| returns 0 if revert is not allowed, otherwise
| revert is allowed
13148|
13149|
13150| This routine returns true in the following cases
13151| 1. Cluster service is not installed
13152| 2. Cluster service is not running
13153| 3. Cluster service is running, but only 1 node
| is active
13154| */
13155|
13156| ULONG ClusterIsRevertAllowed()
13157| {
13158|     BOOL Ret=TRUE;
13159|     __try {
13160|         HMODULE
| Clusapi=LoadLibrary(TEXT("clusapi.dll"));
13161|         tGetNodeClusterState pGetNodeClusterState=NULL;
13162|         ULONG NodesRunning=0;
13163|
13164|         if((Clusapi!=INVALID_HANDLE_VALUE) &&
| (Clusapi!=NULL)) {
13165|             pGetNodeClusterState =
| (tGetNodeClusterState)GetProcAddress(Clusapi,"GetNodeClu
| sterState");
13166|             if(pGetNodeClusterState) {
13167|                 ULONG State=0;
13168|                 pGetNodeClusterState(NULL,&State);
13169|                 DLOG((TEXT("Cluster node state =
| %08x\n"),State));

```

```

13170|  /*
13171|      ClusterStateNotInstalled    The
13172|      | Cluster service is not installed on the node.
13173|      ClusterStateNotConfigured  The Cluster
13174|      | service is installed on the node but has not yet been
13175|      | configured.
13176|      ClusterStateNotRunning     The
13177|      | Cluster service is installed and configured on the node
13178|      | but is not currently running.
13179|      ClusterStateRunning        The
13180|      | Cluster service is installed, configured, and running
13181|      | on the node.
13182|  */
13183|
13184|      if(State!=ClusterStateRunning) {
13185|          // revert is allowed
13186|          Ret = TRUE;
13187|      } else {
13188|          // okay more work is required to
13189|          | determine if revert is
13190|          // allowed
13191|
13192|          NodesRunning =
13193|          | ClusterGetNumberNodesActive();
13194|
13195|          if(NodesRunning>1) {
13196|              // revert not allowed
13197|              Ret = FALSE;
13198|          } else {
13199|              // revert is allowed
13200|              Ret = TRUE;
13201|          }
13202|      } // ClusterStateRunning
13203|  } else {
13204|      DLOG((TEXT("Error, entry point not
13205|      | found for GetNodeClusterState\n")));
13206|  }
13207|      FreeLibrary(Clusapi);
13208|  } else {
13209|      DLOG((TEXT("Error %08x, unable to load
13210|      | clusapi\n"),GetLastError()));
13211|  }
13212|  } __except(EXCEPTION_EXECUTE_HANDLER) {
13213|      DLOG((TEXT("Exception %08x
13214|      | ClusterIsRevertAllowed\n"),GetExceptionCode()));
13215|  }
13216|  return Ret;
13217| }
13218|

```

```

13208| ULONG ClusterIsClusterActive()
13209| {
13210|     BOOL Ret=FALSE;
13211|     __try {
13212|         HMODULE
            | Clusapi=LoadLibrary(TEXT("clusapi.dll"));
13213|         tGetNodeClusterState pGetNodeClusterState=NULL;
13214|         ULONG NodesRunning=0;
13215|
13216|         if((Clusapi!=INVALID_HANDLE_VALUE) &&
            | (Clusapi!=NULL)) {
13217|             pGetNodeClusterState =
            | (tGetNodeClusterState)GetProcAddress(Clusapi,"GetNodeClu
            | sterState");
13218|             if(pGetNodeClusterState) {
13219|                 ULONG State=0;
13220|                 pGetNodeClusterState(NULL,&State);
13221|                 DLOG((TEXT("Cluster node state =
            | %08x\n"),State));
13222|             /*
13223|                 ClusterStateNotInstalled      The
            | Cluster service is not installed on the node.
13224|                 ClusterStateNotConfigured    The Cluster
            | service is installed on the node but has not yet been
            | configured.
13225|                 ClusterStateNotRunning       The
            | Cluster service is installed and configured on the node
            | but is not currently running.
13226|                 ClusterStateRunning          The
            | Cluster service is installed, configured, and running
            | on the node.
13227|             */
13228|
13229|             if(State==ClusterStateRunning) {
13230|                 // revert is allowed
13231|                 Ret = TRUE;
13232|             } else {
13233|                 Ret = FALSE;
13234|             } // ClusterStateRunning
13235|         } else {
13236|             DLOG((TEXT("Error, entry point not
            | found for GetNodeClusterState\n")));
13237|         }
13238|         FreeLibrary(Clusapi);
13239|     } else {
13240|         DLOG((TEXT("Error %08x, unable to load
            | clusapi\n"),GetLastError()));
13241|     }
13242| } __except(EXCEPTION_EXECUTE_HANDLER) {
13243|     DLOG((TEXT("Exception %08x

```

```

    | ClusterIsRevertAllowed\n"),GetExceptionCode()));
13244|    }
13245|    return Ret;
13246| }
13247|
13248| ULONG IsDiskResource( tClusApis Apis, HCLUSTER Cluster,
    | WCHAR *ResourceName)
13249| {
13250|    HRESOURCE Resource =
    | Apis.OpenClusterResource(Cluster,ResourceName);
13251|    ULONG IsDisk = FALSE;
13252|    ULONG Err=0;
13253|    ULONG Returned=0;
13254|
13255|    if(Resource) {
13256|        CLUS_RESOURCE_CLASS_INFO Info;
13257|
13258|        Err = Apis.ClusterResourceControl( Resource,
13259|            NULL,
13260|            CLUSCTL_RESOURCE_GET_CLASS_INFO,
13261|            NULL,0,
13262|            &Info,
13263|            sizeof(Info),
13264|            &Returned
13265|        );
13266|        if(!Err) {
13267|            if(Info.rc == CLUS_RESCLASS_STORAGE) {
13268|                DLOG((TEXT("'%S' is a disk
    | resource\n"),ResourceName));
13269|                IsDisk=TRUE;
13270|                Err = 0;
13271|            } else {
13272|                DLOG((TEXT("'%S' is a not disk
    | resource\n"),ResourceName));
13273|                IsDisk=FALSE;
13274|            }
13275|        } else {
13276|            DLOG((TEXT("Error %08x getting class info
    | for resource '%S'\n"),Err,ResourceName));
13277|        }
13278|        Apis.CloseClusterResource(Resource);
13279|    } else {
13280|        Err = GetLastError();
13281|        DLOG((TEXT("Error %08x opening resource
    | '%S'\n"),Err,ResourceName));
13282|    }
13283|    return IsDisk;
13284| }
13285|
13286|

```

```

13287| ULONG GetDiskTarget( tClusApis Apis, HCLUSTER Cluster,
    | WCHAR *ResourceName, WCHAR *Target)
13288| {
13289|     HRESOURCE Resource =
    | Apis.OpenClusterResource(Cluster,ResourceName);
13290|     ULONG Err=0;
13291|     ULONG Returned=0;
13292|
13293|     if(Resource) {
13294|         CHAR Buffer[65536]={0};
13295|         CLUSPROP_BUFFER_HELPER cbh;
13296|
13297|         cbh.pb = Buffer;
13298|
13299|         Err = Apis.ClusterResourceControl( Resource,
13300|             NULL,
13301|             CLUSCTL_RESOURCE_STORAGE_GET_DISK_INFO,
13302|             NULL,0,
13303|             Buffer,
13304|             sizeof(Buffer),
13305|             &Returned
13306|         );
13307|         if(!Err) {
13308|             while(1) {
13309|                 DLOG((TEXT("Type=%08x,
    | Length=%08x\n"),cbh.pValue->Syntax.dw,cbh.pValue->cbLeng
    | th));
13310|
13311|                 if(cbh.pValue->Syntax.dw ==
    | CLUSPROP_SYNTAX_ENDMARK) {
13312|                     Err = ERROR_NOT_FOUND;
13313|                     break;
13314|                 }
13315|
13316|                 if(cbh.pValue->Syntax.dw ==
    | CLUSPROP_SYNTAX_PARTITION_INFO) {
13317|                     DLOG((TEXT("Volume='%S',
    | label='%S'\n"),
13318|                         | cbh.pPartitionInfoValue->szDeviceName,
13319|                         | cbh.pPartitionInfoValue->szVolumeLabel
13320|                     ));
13321|
    | wcscpy(Target,cbh.pPartitionInfoValue->szDeviceName);
13322|                     if(Target[1]==':') {
13323|                         Target[2]=0;
13324|                     }
13325|                     Err = 0;
13326|                     break;

```

```

13327|         }
13328|
13329|         | cbh.pb+=cbh.pValue->cbLength+sizeof(CLUSPROP_VALUE);
13330|
13331|         | if(cbh.pb-Buffer+sizeof(DWORD)>Returned) {
13332|             Err = ERROR_NOT_FOUND;
13333|             break;
13334|         }
13335|     } else {
13336|         DLOG((TEXT("Error %08x getting class info
13337|         | for resource '%S'\n"),Err,ResourceName));
13338|     }
13339|     Apis.CloseClusterResource(Resource);
13340| } else {
13341|     Err = GetLastError();
13342|     DLOG((TEXT("Error %08x opening resource
13343|     | '%S'\n"),Err,ResourceName));
13344| }
13345| return Err;
13346| }
13347| #define CLUSTER_DISK_ONLINE    0
13348| #define CLUSTER_DISK_OFFLINE   1
13349|
13350| ULONG ClusterChangeDiskStatus( WCHAR *VolumeName, ULONG
13351|     | Status )
13352| {
13353|     ULONG Err;
13354|     tClusApis Apis;
13355|     HCLUSTER Cluster;
13356|     HCLUSENUM Enum;
13357|     HRESOURCE Resource;
13358|     __try {
13359|         Err = ClusterGetApis(&Apis);
13360|         if(!Err) {
13361|             Cluster = Apis.OpenCluster(NULL);
13362|             if(!Cluster) {
13363|                 DLOG((TEXT("unable to open
13364|                 | cluster\n")));
13365|                 Err = GetLastError();
13366|             }
13367|             if(!Err) {
13368|                 Enum =
13369|                 | Apis.ClusterOpenEnum(Cluster,CLUSTER_ENUM_RESOURCE);

```

```

13370|         if(Enum) {
13371|             ULONG Index = 0;
13372|             ULONG Type;
13373|             ULONG Size;
13374|             WCHAR Buffer[64];
13375|
13376|             while(Err==0) {
13377|                 Size = sizeof(Buffer) /
13378| | sizeof(WCHAR); // number of characters not bytes
13379|
13380|                 Err = Apis.ClusterEnum(
13381|                     Enum,
13382|                     Index,
13383|                     &Type,
13384|                     Buffer,
13385|                     &Size
13386|                 );
13387|
13388|                 if(Err==0) {
13389|                     | if(Type==CLUSTER_ENUM_RESOURCE) {
13390|                         DLOG((TEXT("Resource
13391| | '%S' found\n"),Buffer));
13392|
13393|                     | if(IsDiskResource(Apis,Cluster,Buffer)) {
13394|                         WCHAR Target[256];
13395|                         Err =
13396| | GetDiskTarget(Apis,Cluster,Buffer,Target);
13397|                         if(!Err) {
13398|
13399| | if(_wcsicmp(Target,VolumeName)==0) {
13400|
13401| | DLOG((TEXT("Found the resource!!!!!!!!!!\n")));
13402|
13403| | Resource =
13404| | Apis.OpenClusterResource(Cluster,Buffer);
13405|
13406| | if(Resource) {
13407|
13408| | if(Status==CLUSTER_DISK_ONLINE) {
13409|
13410| | Err
13411| | = Apis.OnlineClusterResource(Resource);
13412|
13413| | if(Err) {
13414|
13415| | DLOG((TEXT("Error %08x bring resource online\n"),Err));
13416|
13417| | }
13418| | else
13419|
13420| | if(Status==CLUSTER_DISK_OFFLINE) {

```



```

13406|                                     Err
    | = Apis.OfflineClusterResource(Resource);
13407|
    | if(Err) {
13408|     | DLOG((TEXT("Error %08x bring resource
    | offline\n"),Err));
13409|                                     }
13410|                                     }
13411|
    | DLOG((TEXT("Waiting for device to come
    | online/offline\n")));
13412|
    | while(1) {
13413|     | CLUSTER_RESOURCE_STATE State =
    | Apis.GetClusterResourceState(Resource,NULL,0,NULL,0);
13414|     | if((State==ClusterResourcePending) ||
13415|     | (State==ClusterResourceOnlinePending) ||
13416|     | (State==ClusterResourceOfflinePending)) {
13417|         | DLOG((TEXT("Waiting for device to come
    | online/offline\n")));
13418|         | Sleep(1000);
13419|                                     }
    | else {
13420|         | DLOG((TEXT("Device online/offline\n")));
13421|         | Err = 0;
13422|         | break;
13423|                                     }
13424|
13425| /*
13426| ClusterResourceInitializing The resource is performing
    | initialization.
13427| ClusterResourceOnline The resource is operational and
    | functioning normally.
13428| ClusterResourceOffline The resource is not operational.
13429| ClusterResourceFailed The resource has failed.
13430| ClusterResourcePending The resource is in the process
    | of coming online or going offline.
13431| ClusterResourceOnlinePending The resource is in the
    | process of coming online.
13432| ClusterResourceOfflinePending The resource is in the

```

```

    | process of going offline.
13433| ClusterResourceStateUnknown
13434| */
13435|
13436|
13437|         }
13438|
    | Apis.CloseClusterResource(Resource);
13439|         } else {
13440|         Err =
    | GetLastError();
13441|
    | DLOG((TEXT("Error %08x opening resource\n"),Err));
13442|         }
13443|
13444|         // exit the
    | while loop
13445|         break;
13446|         }
13447|         } else {
13448|
    | DLOG((TEXT("Error getting target of disk resource
    | '%S'\n"),Buffer));
13449|         }
13450|         }
13451|         } else {
13452|         DLOG((TEXT("'%'S' is not
    | a resource\n"),Buffer));
13453|         }
13454|
13455|         Index++;
13456|         }
13457|         } // while(Err==0)
13458|
13459|         Apis.ClusterCloseEnum(Enum);
13460|         } else {
13461|         DLOG((TEXT("Cluster: Error opening
    | enum for resources\n")));
13462|         }
13463|
13464|         Apis.CloseCluster(Cluster);
13465|         } else {
13466|         DLOG((TEXT("Cluster: Error %08x opening
    | cluster\n"),Err));
13467|         }
13468|         FreeLibrary(Apis.ClusApi);
13469|         } else {
13470|         DLOG((TEXT("Cluster: Error %08x unable to
    | get entry points\n"),Err));
13471|         }

```

```

13472| } __except(EXCEPTION_EXECUTE_HANDLER) {
13473|     DLOG((TEXT("Exception %08x
| ClusterGetNumberNodesActive\n"),GetExceptionCode()));
13474| }
13475| return Err;
13476| }
13477|
13478|
13479| ULONG ClusterInitiateRevert( PVOID
| KernelSnapShotPointer )
13480| {
13481|     ULONG Err;
13482|     WCHAR Buffer[65536]={0};
13483|
13484|     Err = Psm_GetKernelSnapShotVolumesW (
13485|         KernelSnapShotPointer,
13486|         Buffer,
13487|         sizeof(Buffer)
13488|     );
13489|
13490|     if(!Err) {
13491|         WCHAR *p=Buffer;
13492|
13493|         // take the volumes offline
13494|         while(*p) {
13495|             // cluster only support drive letters
13496|             if(p[1] == ':') {
13497|                 Err =
| ClusterChangeDiskStatus(p,CLUSTER_DISK_OFFLINE);
13498|                 if(Err==0) {
13499|                     Err =
| ClusterChangeDiskStatus(p,CLUSTER_DISK_ONLINE);
13500|                 }
13501|             }
13502|             p+=wcslen(p)+1;
13503|         }
13504|     } else {
13505|         DLOG((TEXT("Error %08x getting volumes for
| snapshot %08x\n"),KernelSnapShotPointer));
13506|     }
13507|
13508| return Err;
13509| }
13510|
13511|
13512|
13513| File Listing: cluster.h
13514|
13515| typedef LONG
13516| (WINAPI *tClusterRegCreateKey)(

```

```

13517| IN HKEY hKey,
13518| IN LPCWSTR lpszSubKey,
13519| IN DWORD dwOptions,
13520| IN REGSAM samDesired,
13521| IN LPSECURITY_ATTRIBUTES lpSecurityAttributes,
13522| OUT PHKEY phkResult,
13523| OUT OPTIONAL LPDWORD lpdwDisposition
13524| );
13525|
13526| typedef LONG
13527| (WINAPI *tClusterRegOpenKey)(
13528| IN HKEY hKey,
13529| IN LPCWSTR lpszSubKey,
13530| IN REGSAM samDesired,
13531| OUT PHKEY phkResult
13532| );
13533|
13534| typedef LONG
13535| (WINAPI *tClusterRegDeleteKey)(
13536| IN HKEY hKey,
13537| IN LPCWSTR lpszSubKey
13538| );
13539|
13540| typedef LONG
13541| (WINAPI *tClusterRegCloseKey)(
13542| IN HKEY hKey
13543| );
13544|
13545| typedef LONG
13546| (WINAPI *tClusterRegEnumKey)(
13547| IN HKEY hKey,
13548| IN DWORD dwIndex,
13549| OUT LPWSTR lpszName,
13550| IN OUT LPDWORD lpcchName,
13551| OUT PFILETIME lpftLastWriteTime
13552| );
13553|
13554| typedef DWORD
13555| (WINAPI *tClusterRegSetValue)(
13556| IN HKEY hKey,
13557| IN LPCWSTR lpszValueName,
13558| IN DWORD dwType,
13559| IN CONST BYTE* lpData,
13560| IN DWORD cbData
13561| );
13562|
13563| typedef DWORD
13564| (WINAPI *tClusterRegDeleteValue)(
13565| IN HKEY hKey,
13566| IN LPCWSTR lpszValueName

```

```

13567| );
13568|
13569| typedef LONG
13570| (WINAPI *tClusterRegQueryValue)(
13571|     IN HKEY hKey,
13572|     IN LPCWSTR lpszValueName,
13573|     OUT LPDWORD lpdwValueType,
13574|     OUT LPBYTE lpData,
13575|     IN OUT LPDWORD lpcbData
13576| );
13577|
13578| typedef DWORD
13579| (WINAPI *tClusterRegEnumValue)(
13580|     IN HKEY hKey,
13581|     IN DWORD dwIndex,
13582|     OUT LPWSTR lpszValueName,
13583|     IN OUT LPDWORD lpcchValueName,
13584|     OUT LPDWORD lpdwType,
13585|     OUT LPBYTE lpData,
13586|     IN OUT LPDWORD lpcbData
13587| );
13588|
13589| typedef LONG
13590| (WINAPI *tClusterRegQueryInfoKey)(
13591|     IN HKEY hKey,
13592|     IN LPDWORD lpcSubKeys,
13593|     IN LPDWORD lpcchMaxSubKeyLen,
13594|     IN LPDWORD lpcValues,
13595|     IN LPDWORD lpcchMaxValueNameLen,
13596|     IN LPDWORD lpcbMaxValueLen,
13597|     IN LPDWORD lpcbSecurityDescriptor,
13598|     IN PFILETIME lpftLastWriteTime
13599| );
13600|
13601| typedef LONG
13602| (WINAPI *tClusterRegGetKeySecurity )(
13603|     IN HKEY hKey,
13604|     IN SECURITY_INFORMATION RequestedInformation,
13605|     OUT PSECURITY_DESCRIPTOR pSecurityDescriptor,
13606|     IN LPDWORD lpcbSecurityDescriptor
13607| );
13608|
13609| typedef LONG
13610| (WINAPI *tClusterRegSetKeySecurity)(
13611|     IN HKEY hKey,
13612|     IN SECURITY_INFORMATION SecurityInformation,
13613|     IN PSECURITY_DESCRIPTOR pSecurityDescriptor
13614| );
13615|
13616|

```

```

13617| typedef DWORD
13618| (WINAPI *tGetNodeClusterState)(
13619|     IN LPCWSTR lpszNodeName,
13620|     OUT DWORD *pdwClusterState
13621| );
13622|
13623| typedef HCLUSTER
13624| (WINAPI *tOpenCluster)(
13625|     IN LPCWSTR lpszClusterName
13626| );
13627|
13628| typedef BOOL
13629| (WINAPI *tCloseCluster)(
13630|     IN HCLUSTER hCluster
13631| );
13632|
13633| typedef HKEY
13634| (WINAPI *tGetClusterKey)(
13635|     IN HCLUSTER hCluster,
13636|     IN REGSAM samDesired
13637| );
13638|
13639| typedef DWORD (WINAPI *tClusterEnum)(
13640|     HCLUSENUM hEnum,
13641|     DWORD dwIndex,
13642|     LPDWORD lpdwType,
13643|     LPWSTR lpszName,
13644|     LPDWORD lpcchName
13645| );
13646|
13647| typedef HCLUSENUM (WINAPI *tClusterOpenEnum)(
13648|     HCLUSTER hCluster,
13649|     DWORD dwType
13650| );
13651|
13652| typedef DWORD (WINAPI *tClusterCloseEnum)(
13653|     HCLUSENUM hEnum
13654| );
13655|
13656| typedef CLUSTER_NODE_STATE (WINAPI
13657|     | *tGetClusterNodeState)(
13658|     HNODE hNode
13659| );
13660| typedef HNODE (WINAPI *tOpenClusterNode)(
13661|     HCLUSTER hCluster,
13662|     LPCWSTR lpszNodeName
13663| );
13664|
13665| typedef BOOL (WINAPI *tCloseClusterNode)(

```

```

13666| HNODE hNode
13667| );
13668|
13669| typedef DWORD (WINAPI *tResumeClusterNode)(
13670| HNODE hNode
13671| );
13672|
13673| typedef DWORD (WINAPI *tPauseClusterNode)(
13674| HNODE hNode
13675| );
13676|
13677| typedef HRESOURCE (WINAPI *tOpenClusterResource)(
13678| HCLUSTER hCluster,
13679| LPCWSTR lpszResourceName
13680| );
13681|
13682| typedef BOOL (WINAPI *tCloseClusterResource)(
13683| HRESOURCE hResource
13684| );
13685|
13686| typedef DWORD (WINAPI *tOfflineClusterResource)(
13687| HRESOURCE hResource
13688| );
13689|
13690| typedef DWORD (WINAPI *tOnlineClusterResource)(
13691| HRESOURCE hResource
13692| );
13693|
13694| typedef CLUSTER_RESOURCE_STATE (WINAPI
    | *tGetClusterResourceState)(
13695| HRESOURCE hResource,
13696| LPWSTR lpszNodeName,
13697| LPDWORD lpcchNodeName,
13698| LPWSTR lpszGroupName,
13699| LPDWORD lpcchGroupName
13700| );
13701|
13702| typedef DWORD (WINAPI *tClusterResourceControl)(
13703| HRESOURCE hResource,
13704| HNODE hHostNode,
13705| DWORD dwControlCode,
13706| LPVOID lpInBuffer,
13707| DWORD cbInBufferSize,
13708| LPVOID lpOutBuffer,
13709| DWORD cbOutBufferSize,
13710| LPDWORD lpcbBytesReturned
13711| );
13712|
13713| typedef struct sClusApis {
13714| HMODULE ClusApi;

```

```

13715| tOpenCluster OpenCluster;
13716| tCloseCluster CloseCluster;
13717| tGetClusterKey GetClusterKey;
13718| tClusterRegCreateKey ClusterRegCreateKey;
13719| tClusterRegOpenKey ClusterRegOpenKey;
13720| tClusterRegDeleteKey ClusterRegDeleteKey;
13721| tClusterRegCloseKey ClusterRegCloseKey;
13722| tClusterRegEnumKey ClusterRegEnumKey;
13723| tClusterRegSetValue ClusterRegSetValue;
13724| tClusterRegDeleteValue ClusterRegDeleteValue;
13725| tClusterRegQueryValue ClusterRegQueryValue;
13726| tClusterRegEnumValue ClusterRegEnumValue;
13727| tClusterRegQueryInfoKey ClusterRegQueryInfoKey;
13728| tClusterRegGetKeySecurity ClusterRegGetKeySecurity;
13729| tClusterRegSetKeySecurity ClusterRegSetKeySecurity;
13730| tClusterOpenEnum ClusterOpenEnum;
13731| tClusterEnum ClusterEnum;
13732| tClusterCloseEnum ClusterCloseEnum;
13733| tGetNodeClusterState GetNodeClusterState;
13734| tGetClusterNodeState GetClusterNodeState;
13735| tOpenClusterNode OpenClusterNode;
13736| tCloseClusterNode CloseClusterNode;
13737| tResumeClusterNode ResumeClusterNode;
13738| tPauseClusterNode PauseClusterNode;
13739| tOpenClusterResource OpenClusterResource;
13740| tCloseClusterResource CloseClusterResource;
13741| tOfflineClusterResource OfflineClusterResource;
13742| tOnlineClusterResource OnlineClusterResource;
13743| tGetClusterResourceState GetClusterResourceState;
13744| tClusterResourceControl ClusterResourceControl;
13745|
13746| } tClusApis,*pClusApis;
13747|
13748| ULONG UpdateClusterRegistries( tSnapShot *SnapShot );
13749| ULONG IsClusterInstalled(void);
13750| ULONG ClusterUpdateVolume( WCHAR *NTName );
13751| ULONG ClusterIsRevertAllowed();
13752| ULONG ClusterIsClusterActive();
13753| ULONG ClusterInitiateRevert( PVOID
    | KernelSnapShotPointer );
13754|
13755|
13756|
13757| File Listing: dasd.c
13758|
13759| #include <stdio.h>
13760| #include <stdlib.h>
13761| #include <string.h>
13762| #include <windows.h>
13763| #include <windowsx.h>

```



```

13764| #include <winioctl.h>
13765| #include <tchar.h>
13766| #include <conio.h>
13767| #include "mytypes.h"
13768| #include "dasd.h"
13769| #include "dlog.h"
13770|
13771| #include <undoc.h>
13772|
13773| //void PSM_LogDebugInfo( const TCHAR *fmt,...);
13774|
13775| /*
13776|    This code was taken from spti.c from the ddk
    | samples.
13777|
13778|    It works on the fact that a buffer that is aligned
    | on a
13779|    cetain boundary will have 0's in its least
    | significant bits
13780|    depending on the alignment requirements, A pointer
    | is
13781|    manipulated so that it is pointing to the first
    | possible
13782|    address that meets the alignment requirements.
13783| */
13784| void *AllocBufferBelow16Meg( size_t size, ULONG Align )
13785| {
13786|    PCHAR ptr;
13787|
13788|    if (!Align) {
13789|        ptr = (char*)malloc(size);
13790|    } else {
13791|        ptr = (char*)malloc(size+Align);
13792|        if (ptr)
13793|            ptr = (PCHAR)((((ULONG)ptr+Align) & ~Align);
13794|    }
13795|    return ptr;
13796| }
13797|
13798| void FreeBufferBelow16Meg( void *Buffer )
13799| {
13800|    free(Buffer);
13801|    return;
13802| }
13803|
13804| ULONG DASD_DeviceCount( void )
13805| {
13806|    TCHAR *Buffer,*p;
13807|    LONG Count=0;
13808|

```

```

13809|
13810| Buffer = (TCHAR*)malloc(65536*sizeof(TCHAR));
13811| QueryDosDevice(NULL,Buffer,65536);
13812|
13813| p = Buffer;
13814| while(*p) {
13815|     if(_tcsnicmp(TEXT("PhysicalDrive"),p,13)==0) {
13816|         Count++;
13817|     }
13818|     p = p + lstrlen(p)+1;
13819| }
13820|
13821|
13822| free(Buffer);
13823| return Count;
13824| }
13825|
13826|
13827| ULONG DASD_GetDriveGeometry( HANDLE hDevice,
    | DISK_GEOMETRY *geometry )
13828| {
13829|     ULONG returned;
13830|
13831|     return DeviceIoControl( hDevice,
13832|         IOCTL_DISK_GET_DRIVE_GEOMETRY,
13833|         NULL,
13834|         0,
13835|         geometry,
13836|         sizeof(DISK_GEOMETRY),
13837|         &returned,
13838|         FALSE);
13839| }
13840|
13841| ULONG DASD_GetDriveGeometry2( HANDLE hDevice,
    | DISK_GEOMETRY *geometry )
13842| {
13843|     ULONG returned;
13844|     DISK_GEOMETRY Geos[10];
13845|     ULONG OK;
13846|
13847|     OK = DeviceIoControl( hDevice,
13848|         IOCTL_DISK_GET_MEDIA_TYPES,
13849|         NULL,
13850|         0,
13851|         &Geos,
13852|         sizeof(Geos),
13853|         &returned,
13854|         FALSE);
13855|     if (OK) {
13856|

```

```

    | memmove(geometry,&Geos[0],sizeof(DISK_GEOMETRY));
13857| }
13858|
13859| return OK;
13860| }
13861|
13862| tDASDHandle *DASD_LockDevice( ULONG DeviceNum )
13863| {
13864|     tDASDHandle *Packet=NULL;
13865|     TCHAR Name[40];
13866|     HANDLE hDevice;
13867|     DISK_GEOMETRY geometry;
13868|     ULONG GeometryOK = FALSE;
13869|
13870|     | _sprintf(Name,TEXT("\\\\.\\PhysicalDrive%d"),DeviceNum)
    | ;
13871|     hDevice = CreateFile( Name,
13872|         GENERIC_READ | GENERIC_WRITE,
13873|         FILE_SHARE_READ | FILE_SHARE_WRITE,
13874|         NULL,
13875|         OPEN_EXISTING,
13876|         FILE_ATTRIBUTE_NORMAL,
13877|         NULL
13878|     );
13879|
13880|     if(hDevice) {
13881|
13882|         GeometryOK = DASD_GetDriveGeometry( hDevice,
    | &geometry );
13883|
13884|         Packet =
    | (tDASDHandle*)malloc(sizeof(tDASDHandle));
13885|         if (Packet) {
13886|             Packet->Buffer =
    | (BYTE*)AllocBufferBelow16Meg(DEFAULTBUFFERSIZE,0);
13887|             if(Packet->Buffer) {
13888|                 Packet->DasdType      = DASD;
13889|                 Packet->DeviceNum      =
    | LONGTOBYTE(DeviceNum);
13890|                 Packet->hDevice        = hDevice;
13891|                 Packet->Self           =
    | (tDevice*)malloc(sizeof(tDevice));
13892|                 if(GeometryOK) {
13893|                     Packet->Self->BlockSize =
    | geometry.BytesPerSector;
13894|                     Packet->Self->CylindersHigh =
    | geometry.Cylinders.HighPart;
13895|                     Packet->Self->CylindersLow =
    | geometry.Cylinders.LowPart;

```

```

13896|         Packet->Self->Heads      =
| geometry.TracksPerCylinder;
13897|         Packet->Self->SPT        =
| geometry.SectorsPerTrack;
13898|         Packet->Self->Removable   =
| geometry.MediaType == RemovableMedia ? 1 : 0;
13899|         Packet->Self->NumSectors  =
| geometry.Cylinders.LowPart * geometry.TracksPerCylinder
| * geometry.SectorsPerTrack;
13900|     } else {
13901|         Packet->Self->BlockSize   = 512;
13902|         Packet->Self->CylindersHigh = 0;
13903|         Packet->Self->CylindersLow  = 0;
13904|         Packet->Self->Heads        = 64;
13905|         Packet->Self->SPT          = 32;
13906|         Packet->Self->Removable     = 1;
13907|         Packet->Self->NumSectors    = 0;
13908|     }
13909| } else {
13910|     free(Packet);
13911|     Packet=NULL;
13912| }
13913| }
13914| }
13915| return Packet;
13916| }
13917|
13918| tDASDHandle *DASD_LockVolume( WCHAR *NTDeviceName)
13919| {
13920|     tDASDHandle *Packet=NULL;
13921|     HANDLE hDevice = INVALID_HANDLE_VALUE;
13922|     DISK_GEOMETRY geometry;
13923|     ULONG GeometryOK = FALSE;
13924|
13925|     ULONG Err=0;
13926|     ULONG OldMode;
13927|     UNICODE_STRING Uni={0};
13928|     OBJECT_ATTRIBUTES ObjectAttributes={0};
13929|     IO_STATUS_BLOCK IoStatus={0};
13930|     ULONG Access;
13931|     WCHAR *Name;
13932|
13933|     OldMode = SetErrorMode(SEM_FAILCRITICALERRORS);
13934|
13935|     Name =
| LocalAlloc(LPTR,wcslen(NTDeviceName)*sizeof(WCHAR)+2*siz
| eof(WCHAR));
13936|     if(!Name) {
13937|         return NULL;
13938|     }

```

```

13939|
13940| // we need to "root" object
13941| swprintf(Name,L"%s",NTDeviceName);
13942|
13943| RtlInitUnicodeString( &Uni, Name);
13944|
13945| InitializeObjectAttributes ( &ObjectAttributes,
13946|                             &Uni,
13947|                             OBJ_CASE_INSENSITIVE,
13948|                             NULL,
13949|                             NULL );
13950|
13951|
13952| Access = FILE_GENERIC_WRITE | FILE_GENERIC_READ;
13953|
13954|
13955| DoOpen:
13956| Err = NtCreateFile( &hDevice,
13957|                   Access,          // desired
13958|                   | access
13959|                   &ObjectAttributes,    //
13960|                   | object attributes
13961|                   &IoStatus,
13962|                   NULL,          // alloc size
13963|                   FILE_ATTRIBUTE_NORMAL,    // file
13964|                   | attributes
13965|                   FILE_SHARE_WRITE | FILE_SHARE_READ,
13966|                   | // share access
13967|                   FILE_OPEN,      // create
13968|                   | disposition
13969|                   FILE_SYNCHRONOUS_IO_NONALERT,
13970|                   | // create options
13971|                   NULL, // eabuffer
13972|                   0 ); // ealength
13973| if(Err) {
13974|     if(Access & FILE_WRITE_DATA) {
13975|         Access = FILE_GENERIC_READ;
13976|     } else
13977|     if(Access & FILE_READ_DATA) {
13978|         Access = 0;
13979|     } else {
13980|         Err = GetLastError();
13981|         goto Done;
13982|     }
13983|     goto DoOpen;
13984| }
13985| Done:
13986| LocalFree(Name);
13987|
13988| SetErrorMode(OldMode);

```

```

13983|
13984|   if(hDevice!=INVALID_HANDLE_VALUE) {
13985|
13986|       GeometryOK = DASD_GetDriveGeometry( hDevice,
        | &geometry );
13987|
13988|       Packet =
        | (tDASDHandle*)malloc(sizeof(tDASDHandle));
13989|       if (Packet) {
13990|           Packet->Buffer =
        | (BYTE*)AllocBufferBelow16Meg(DEFAULTBUFFERSIZE,0);
13991|           if(Packet->Buffer) {
13992|               Packet->DasdType      = VOLUME;
13993|               Packet->DeviceNum      = -1;
13994|               Packet->hDevice        = hDevice;
13995|               Packet->Self           =
        | (tDevice*)malloc(sizeof(tDevice));
13996|               if(GeometryOK) {
13997|                   Packet->Self->BlockSize  =
        | geometry.BytesPerSector;
13998|                   Packet->Self->CylindersHigh =
        | geometry.Cylinders.HighPart;
13999|                   Packet->Self->CylindersLow  =
        | geometry.Cylinders.LowPart;
14000|                   Packet->Self->Heads        =
        | geometry.TracksPerCylinder;
14001|                   Packet->Self->SPT          =
        | geometry.SectorsPerTrack;
14002|                   Packet->Self->Removable    =
        | geometry.MediaType == RemovableMedia ? 1 : 0;
14003|                   Packet->Self->NumSectors  =
        | geometry.Cylinders.LowPart * geometry.TracksPerCylinder
        | * geometry.SectorsPerTrack;
14004|               } else {
14005|                   Packet->Self->BlockSize    = 512;
14006|                   Packet->Self->CylindersHigh = 0;
14007|                   Packet->Self->CylindersLow  = 0;
14008|                   Packet->Self->Heads         = 64;
14009|                   Packet->Self->SPT           = 32;
14010|                   Packet->Self->Removable     = 1;
14011|                   Packet->Self->NumSectors    = 0;
14012|               }
14013|           } else {
14014|               free(Packet);
14015|               Packet=NULL;
14016|           }
14017|       }
14018|   }
14019|   return Packet;
14020| }

```

```

14021|
14022|
14023|
14024| /*
14025|   Opens the device in passive mode (No Error boxes
      | pop up.
14026|   No Reads or writes are allowed to a device that is
      | opened this
14027|   way.
14028| */
14029| tDASDHandle *DASD_PassiveLockDevice( ULONG DeviceNum )
14030| {
14031|   tDASDHandle *Packet=NULL;
14032|   TCHAR Name[40];
14033|   HANDLE hDevice = INVALID_HANDLE_VALUE;
14034|   DISK_GEOMETRY geometry;
14035|   ULONG GeometryOK = FALSE;
14036|
14037|   |_sprintf(Name,TEXT("\\\\.\\PhysicalDrive%d"),DeviceNum)
      | ;
14038|   hDevice = CreateFile( Name,
14039|       0,
14040|       FILE_SHARE_READ | FILE_SHARE_WRITE,
14041|       NULL,
14042|       OPEN_EXISTING,
14043|       FILE_ATTRIBUTE_NORMAL,
14044|       NULL
14045|   );
14046|
14047|   if(hDevice) {
14048|
14049|       GeometryOK = DASD_GetDriveGeometry( hDevice,
      | &geometry );
14050|
14051|       Packet =
      | (tDASDHandle*)malloc(sizeof(tDASDHandle));
14052|       if (Packet) {
14053|           Packet->Buffer =
      | (BYTE*)AllocBufferBelow16Meg(DEFAULTBUFFERSIZE,0);
14054|           if(Packet->Buffer) {
14055|               Packet->DasdType      = DASD;
14056|               Packet->DeviceNum      =
      | LONGTOBYTE(DeviceNum);
14057|               Packet->hDevice        = hDevice;
14058|               Packet->Self            =
      | (tDevice*)malloc(sizeof(tDevice));
14059|               if(GeometryOK) {
14060|                   Packet->Self->BlockSize =
      | geometry.BytesPerSector;

```

```

14061|          Packet->Self->CylindersHigh =
| geometry.Cylinders.HighPart;
14062|          Packet->Self->CylindersLow  =
| geometry.Cylinders.LowPart;
14063|          Packet->Self->Heads      =
| geometry.TracksPerCylinder;
14064|          Packet->Self->SPT        =
| geometry.SectorsPerTrack;
14065|          Packet->Self->Removable   =
| geometry.MediaType == RemovableMedia ? 1 : 0;
14066|          Packet->Self->NumSectors  =
| geometry.Cylinders.LowPart * geometry.TracksPerCylinder
| * geometry.SectorsPerTrack;
14067|      } else {
14068|
14069|          Packet->Self->BlockSize   = 512;
14070|          Packet->Self->CylindersHigh = 0;
14071|          Packet->Self->CylindersLow  = 0;
14072|          Packet->Self->Heads        = 64;
14073|          Packet->Self->SPT          = 32;
14074|          Packet->Self->Removable     = 1;
14075|          Packet->Self->NumSectors    = 0;
14076|      }
14077|  } else {
14078|      free(Packet);
14079|      Packet=NULL;
14080|  }
14081|  }
14082|  }
14083|  return Packet;
14084| }
14085|
14086| ULONG DASD_UnlockDevice( tDASDHandle *LockHandle )
14087| {
14088|     if (LockHandle) {
14089|         if(LockHandle->hDevice!=(HANDLE)-1) {
14090|             if(LockHandle->DasdType == VOLUME) {
14091|                 NtClose(LockHandle->hDevice);
14092|             } else {
14093|                 CloseHandle(LockHandle->hDevice);
14094|             }
14095|         }
14096|         if(LockHandle->Buffer) {
14097|             FreeBufferBelow16Meg(LockHandle->Buffer);
14098|         }
14099|         if(LockHandle->Self) {
14100|             free(LockHandle->Self);
14101|         }
14102|         free(LockHandle);
14103|     }

```



```

14104|    return 0;
14105| }
14106|
14107| ULONG DASD_Read(
14108|         tDASDHandle *LockHandle,
14109|         ULONGLONG Sector,
14110|         ULONG Count,
14111|         void *Buffer
14112|     )
14113| {
14114|     ULONG NumberOfBytesRead;
14115|     ULONG res;
14116|     ULARGE_INTEGER BytePos;
14117|     //LONGLONG *b=(LONGLONG*)&BytePos;
14118|
14119|     __try {
14120|         //*b = Int32x32To64
14121|         | (Sector,LockHandle->Self->BlockSize);
14122|         BytePos.QuadPart = Sector *
14123|         | LockHandle->Self->BlockSize;
14124|
14125|         if(SetFilePointer(
14126|             LockHandle->hDevice,    // handle of file
14127|             BytePos.LowPart,        // number of
14128|             | bytes to move file pointer
14129|             &BytePos.HighPart,     // address of
14130|             | high-order word of distance to move
14131|             FILE_BEGIN // how to move
14132|         )!=0xffffffff) {
14133|             Good_Read:
14134|             if (ReadFile(
14135|                 LockHandle->hDevice, // handle of
14136|                 | file to read
14137|                 Buffer, // address of buffer that
14138|                 | receives data
14139|                 Count*LockHandle->Self->BlockSize, //
14140|                 | number of bytes to read
14141|                 &NumberOfBytesRead, // address of
14142|                 | number of bytes read
14143|                 NULL // address of structure for data
14144|             )==TRUE) {
14145|                 res = 0;
14146|             } else {
14147|                 res = GetLastError();
14148|             }
14149|         } else {
14150|             res = GetLastError();
14151|             if(res==NO_ERROR)
14152|                 goto Good_Read;
14153|         }
14154|     }

```

```

14146| } __except(EXCEPTION_EXECUTE_HANDLER) {
14147|     res = GetExceptionCode();
14148|     DLOG((TEXT("Exception %08x while reading device
    | %p, Sector %d, %d,
    | Buffer=%p"),res,LockHandle,Sector,Count,Buffer));
14149| }
14150|
14151|
14152| return res;
14153| }
14154|
14155| ULONG DASD_Write(
14156|         tDASDHandle *LockHandle,
14157|         ULONGLONG Sector,
14158|         ULONG Count,
14159|         void *Buffer
14160|     )
14161| {
14162| #if 0
14163|     return 0;
14164| #else
14165|     ULONG NumberOfBytesWritten;
14166|     ULONG res;
14167|     ULARGE_INTEGER BytePos;
14168|     //LONGLONG *b=(LONGLONG*)&BytePos;
14169|
14170|     __try {
14171|         /*b = Int32x32To64
    | (Sector,LockHandle->Self->BlockSize);
14172|         BytePos.QuadPart = Sector *
    | LockHandle->Self->BlockSize;
14173|
14174|         if(SetFilePointer(
14175|             LockHandle->hDevice, // handle of file
14176|             BytePos.LowPart, // number of
    | bytes to move file pointer
14177|             &BytePos.HighPart, // address of
    | high-order word of distance to move
14178|             FILE_BEGIN // how to move
14179|         )!=0xffffffff) {
14180|             Good_Write:
14181|             if (WriteFile(
14182|                 LockHandle->hDevice, // handle of
    | file to read
14183|                 Buffer, // address of buffer that
    | receives data
14184|                 Count*LockHandle->Self->BlockSize, //
    | number of bytes to read
14185|                 &NumberOfBytesWritten, // address of
    | number of bytes read

```

```

14186|         NULL // address of structure for data
14187|     )==TRUE) {
14188|         res = 0;
14189|     } else {
14190|         res = GetLastError();
14191|     }
14192| } else {
14193|     res = GetLastError();
14194|     if(res==NO_ERROR)
14195|         goto Good_Write;
14196| }
14197| } __except(EXCEPTION_EXECUTE_HANDLER) {
14198|     res = GetExceptionCode();
14199|     DLOG((TEXT("Exception %08x while writing device
        | %p, Sector %d, %d,
        | Buffer=%p"),res,LockHandle,Sector,Count,Buffer));
14200| }
14201|
14202|
14203|     return res;
14204| #endif
14205| }
14206|
14207|
14208|
14209| File Listing: dasd.h
14210|
14211| #define MAXDEVICES 32
14212| #define DEFAULTBUFFERSIZE 32768
14213| #define DASD 1
14214| #define VOLUME 2
14215|
14216| typedef struct sObject {
14217|     ULONG     Type;
14218|     ULONG     StartBlock;
14219|     ULONG     EndBlock;
14220|     string8    Name[20];
14221| } tObject;
14222|
14223| typedef struct sDevice {
14224|     string8    Name[60];    // 60
14225|     ULONG     MaxIOSize;    // 4
14226|     ULONG     BlockSize;    // 4
14227|     ULONG     DriveType;    // 4
14228|     ULONG     Removable;
14229|     ULONG     NumSectors;    // 4
14230|     ULONG     DriveID;       // 4
14231|     ULONG     NumObjects;    // 4
14232|     tObject    Objects[16];  // 512
        | (20+4+4+4)*16

```

```

14233|  BYTE      ReservedBig[640];
14234|
14235|  // _NT_ and RPC
14236|
14237|  ULONG CylindersLow;
14238|  ULONG CylindersHigh;
14239|
14240|  ULONG Heads;    // 4
14241|  ULONG SPT;      // 4
14242|  ULONG Reserved5;    // 4
14243|  ULONG Writes;    // 4
14244|  ULONG Reads;     // 4
14245|  ULONG WriteSecs;    // 4
14246|  ULONG ReadSecs;    // 4
14247|  ULONG IOCount;    // 4
14248|  ULONG IOCTLCount;    // 4
14249|
14250| } tDevice,*pDevice;          //1164  * 32 = 37372
14251|
14252|
14253| typedef struct sDASDHandle {
14254|  ULONG DasdType;
14255|  BYTE DeviceNum;
14256|  HANDLE hDevice;
14257|  unsigned char *Buffer;
14258|  tDevice *Self;
14259| } tDASDHandle,*pDASDHandle;
14260|
14261| extern tDevice  c_DeviceList[];
14262| extern ULONG    DeviceCount;
14263|
14264| void *AllocBufferBelow16Meg( size_t size, ULONG Align
    | );
14265| void FreeBufferBelow16Meg( void *Buffer );
14266| ULONG DASD_DeviceCount( void );
14267| ULONG DASD_GetDriveGeometry( HANDLE hDevice,
    | DISK_GEOMETRY *geometry );
14268| ULONG DASD_GetDriveGeometry2( HANDLE hDevice,
    | DISK_GEOMETRY *geometry );
14269| tDASDHandle *DASD_LockDevice( ULONG DeviceNum );
14270| tDASDHandle *DASD_PassiveLockDevice( ULONG DeviceNum );
14271| ULONG DASD_UnlockDevice( tDASDHandle *LockHandle );
14272| ULONG DASD_Read(
14273|     tDASDHandle *LockHandle,
14274|     ULONGLONG Sector,
14275|     ULONG Count,
14276|     void *Buffer
14277| );
14278| ULONG DASD_Write(
14279|     tDASDHandle *LockHandle,

```

```

14280|          ULONGLONG Sector,
14281|          ULONG Count,
14282|          void *Buffer
14283|      );
14284| tDASDHandle *DASD_LockVolume( WCHAR *VolumeName );
14285|
14286|
14287|
14288| File Listing: DEFRAG.c
14289|
14290| #include <stdio.h>
14291| #include <stdlib.h>
14292| #include <stdarg.h>
14293| #include <string.h>
14294| #include <time.h>
14295| #include <process.h>
14296| #include <io.h>
14297| #include <errno.h>
14298| #include <conio.h>
14299| #include <fcntl.h>
14300| #include <windows.h>
14301| #include <windowsx.h>
14302| #include <commctrl.h> // includes the common control
    | header
14303| #include <tchar.h>
14304| #include <winioctl.h>
14305| #include <ntddscsi.h>
14306| #include <lm.h>
14307| #include <tchar.h>
14308|
14309| #include <undoc.h>
14310| #include <psm.h>
14311| #include "..\driver\ioctl.h"
14312|
14313| #ifdef _DEBUG
14314| extern ULONG DebugMode;
14315|
14316| #define DLOG(x) if(DebugMode) _tprintf x
14317| #else
14318| #define DLOG(x)
14319| #endif
14320| #define VOLUME_DEFRAG    0x80000000
14321|
14322| #define STATUS_SUCCESS
    | ((NTSTATUS)0x00000000L)
14323| #define STATUS_BUFFER_OVERFLOW
    | ((NTSTATUS)0x80000005L)
14324| #define STATUS_INVALID_PARAMETER
    | ((NTSTATUS)0xC000000DL)
14325| #define STATUS_BUFFER_TOO_SMALL

```

```

| ((NTSTATUS)0xC0000023L)
14326| #define STATUS_ALREADY_COMMITTED
| ((NTSTATUS)0xC0000021L)
14327| #define STATUS_INVALID_DEVICE_REQUEST
| ((NTSTATUS)0xC0000010L)
14328|
14329| //
14330| // Apc Routine (see NTDDK.H)
14331| //
14332| typedef VOID (*PIO_APC_ROUTINE) (
14333|     PVOID ApcContext,
14334|     PIO_STATUS_BLOCK IoStatusBlock,
14335|     ULONG Reserved
14336| );
14337| //
14338| // The undocumented NtFsControlFile
14339| //
14340| // This function is used to send File System Control
| (FSCTL)
14341| // commands into file system drivers. Its definition is
14342| // in ntdll.dll (ntdll.lib), a file shipped with the
| NTDDK.
14343| //
14344| NTSTATUS WINAPI NtFsControlFile(
14345|     HANDLE FileHandle,
14346|     HANDLE Event,           //
| optional
14347|     PIO_APC_ROUTINE ApcRoutine, //
| optional
14348|     PVOID ApcContext,       //
| optional
14349|     PIO_STATUS_BLOCK IoStatusBlock,
14350|     ULONG FsControlCode,
14351|     PVOID InputBuffer,      //
| optional
14352|     ULONG InputBufferLength,
14353|     PVOID OutputBuffer,     //
| optional
14354|     ULONG OutputBufferLength
14355| );
14356|
14357| /*
14358| #define FSCTL_GET_NTFS_FILE_RECORD
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 26, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // NTFS_FILE_RECORD_INPUT_BUFFER,
| NTFS_FILE_RECORD_OUTPUT_BUFFER
14359| */
14360| ULONG FS_GetNTFSFileRecord( HANDLE Handle,
| NTFS_FILE_RECORD_INPUT_BUFFER *NFRI,
| NTFS_FILE_RECORD_OUTPUT_BUFFER *NFRO, ULONG NFROSize )

```

```

14361| {
14362|     ULONG Err;
14363|     IO_STATUS_BLOCK IoStatusBlock={0};
14364|
14365|     __try {
14366|         Err = NtFsControlFile(
14367|             Handle,
14368|             NULL,          //
14369|             | event optional
14370|             NULL,          // APC Routine
14371|             | optional
14372|             NULL,          // APC
14373|             | Context optional
14374|             &IoStatusBlock,
14375|             FSCTL_GET_NTFS_FILE_RECORD,
14376|             NFRI,          // optional
14377|             | sizeof(NTFS_FILE_RECORD_INPUT_BUFFER),
14378|             NFRO,          // optional
14379|             NFROSize
14380|         );
14381|         if(Er==STATUS_PENDING) {
14382|             WaitForSingleObject( Handle, INFINITE );
14383|             Err = IoStatusBlock.Status;
14384|         }
14385|         printf("GetNTFSFileRecord = %08x, %08x,
14386|             | %08x\n",Err,IoStatusBlock.Status,IoStatusBlock.Informati
14387|             | on);
14388|     } __except(EXCEPTION_EXECUTE_HANDLER) {
14389|         Err = GetExceptionCode();
14390|         printf("GetNTFSFileRecord Exception
14391|             | %08x\n",Err);
14392|     }
14393|     return Err;
14394| }
14395|
14396| /*
14397| #define FSCTL_GET_NTFS_VOLUME_DATA
14398|     | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 25, METHOD_BUFFERED,
14399|     | FILE_ANY_ACCESS) // , NTFS_VOLUME_DATA_BUFFER
14400| */
14401| ULONG FS_GetNTFSVolumeData( HANDLE Handle,
14402|     | NTFS_VOLUME_DATA_BUFFER *NVD )
14403| {
14404|     ULONG Err;
14405|     IO_STATUS_BLOCK IoStatusBlock={0};
14406|
14407|     __try {
14408|         Err = NtFsControlFile(
14409|             Handle,

```

```

14401|          NULL,          //
      | event optional
14402|          NULL,          // APC Routine
      | optional
14403|          NULL,          // APC
      | Context optional
14404|          &IoStatusBlock,
14405|          FSCTL_GET_NTFS_VOLUME_DATA,
14406|          NULL,          // optional
14407|          0,
14408|          NVD,          // optional
14409|          sizeof(NTFS_VOLUME_DATA_BUFFER)
14410|      );
14411|      if(Err==STATUS_PENDING) {
14412|          WaitForSingleObject( Handle, INFINITE );
14413|          Err = IoStatusBlock.Status;
14414|      }
14415|      printf("GetNTFSVolumeData = %08x, %08x,
      | %08x\n",Err,IoStatusBlock.Status,IoStatusBlock.Informati
      | on);
14416|  } __except(EXCEPTION_EXECUTE_HANDLER) {
14417|      Err = GetExceptionCode();
14418|      printf("GetNTFSVolumeData Exception
      | %08x\n",Err);
14419|  }
14420|  return Err;
14421| }
14422|
14423| /*
14424| #define FSCTL_GET_VOLUME_BITMAP
      | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 27, METHOD_NEITHER,
      | FILE_ANY_ACCESS) // STARTING_LCN_INPUT_BUFFER,
      | VOLUME_BITMAP_BUFFER
14425| */
14426| ULONG FS_GetVolumeBitmap( HANDLE Handle,
      | STARTING_VCN_INPUT_BUFFER *SVIB, VOLUME_BITMAP_BUFFER
      | *VB, ULONG VBSize )
14427| {
14428|     ULONG Err;
14429|     IO_STATUS_BLOCK IoStatusBlock={0};
14430|
14431|     __try {
14432|         Err = NtFsControlFile(
14433|             Handle,
14434|             NULL,          //
      | event optional
14435|             NULL,          // APC Routine
      | optional
14436|             NULL,          // APC
      | Context optional

```



```

14437|          &IoStatusBlock,
14438|          FSCTL_GET_VOLUME_BITMAP,
14439|          SVIB,          // optional
14440|
14441|          | sizeof(STARTING_VCN_INPUT_BUFFER),
14442|          VB,          // optional
14443|          VBSize
14444|      );
14445|      if(Err==STATUS_PENDING) {
14446|          WaitForSingleObject( Handle, INFINITE );
14447|          Err = IoStatusBlock.Status;
14448|      }
14449|      //printf("GetVolumeBitmap= %08x, %08x,
14450|          | %08x\n",Err,IoStatusBlock.Status,IoStatusBlock.Informati
14451|          | on);
14452|      } __except(EXCEPTION_EXECUTE_HANDLER) {
14453|          Err = GetExceptionCode();
14454|          printf("GetVolumeBitmap Exception %08x\n",Err);
14455|      }
14456|      return Err;
14457| }
14458|
14459| /*
14460| #define FSCTL_GET_RETRIEVAL_POINTERS
14461|      | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 28, METHOD_NEITHER,
14462|      | FILE_ANY_ACCESS) // STARTING_VCN_INPUT_BUFFER,
14463|      | RETRIEVAL_POINTERS_BUFFER
14464| */
14465|
14466| ULONG FS_GetRetrievalPointers( HANDLE Handle,
14467|      | STARTING_VCN_INPUT_BUFFER *SVIB,
14468|      | RETRIEVAL_POINTERS_BUFFER *RP, ULONG RPSize )
14469| {
14470|     ULONG Err;
14471|     IO_STATUS_BLOCK IoStatusBlock={0};
14472|
14473|     __try {
14474|         Err = NtFsControlFile(
14475|             Handle,
14476|             NULL,          //
14477|             | event optional
14478|             NULL,          // APC Routine
14479|             | optional
14480|             NULL,          // APC
14481|             | Context optional
14482|             &IoStatusBlock,
14483|             FSCTL_GET_RETRIEVAL_POINTERS,
14484|             SVIB,          // optional
14485|             | sizeof(STARTING_VCN_INPUT_BUFFER),

```

```

14475|         RP,          // optional
14476|         RPSize
14477|     );
14478|     if(Err==STATUS_PENDING) {
14479|         WaitForSingleObject( Handle, INFINITE );
14480|         Err = IoStatusBlock.Status;
14481|     }
14482|     //printf("GetRetrievalPointer= %08x, %08x,
| %08x\n",Err,IoStatusBlock.Status,IoStatusBlock.Informati
| on);
14483| } __except(EXCEPTION_EXECUTE_HANDLER) {
14484|     Err = GetExceptionCode();
14485|     printf("GetRetrievalPointer Exception
| %08x\n",Err);
14486| }
14487| return Err;
14488| }
14489|
14490| /*
14491| #define FSCTL_MOVE_FILE
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 29, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // MOVE_FILE_DATA,
14492| */
14493|
14494| ULONG FS_MoveFile( HANDLE Handle, MOVE_FILE_DATA *MF )
14495| {
14496|     ULONG Err;
14497|     IO_STATUS_BLOCK IoStatusBlock={0};
14498|
14499|     __try {
14500|         Err = NtFsControlFile(
14501|             Handle,
14502|             NULL,          //
| event optional
14503|             NULL,          // APC Routine
| optional
14504|             NULL,          // APC
| Context optional
14505|             &IoStatusBlock,
14506|             FSCTL_MOVE_FILE,
14507|             MF,            // optional
14508|             sizeof(MOVE_FILE_DATA),
14509|             NULL,          // optional
14510|             0
14511|         );
14512|         if(Err==STATUS_PENDING) {
14513|             WaitForSingleObject( Handle, INFINITE );
14514|             Err = IoStatusBlock.Status;
14515|         }
14516|         printf("MoveFile = %08x, %08x,

```

```

    | %08x\n",Err,loStatusBlock.Status,loStatusBlock.Informati
    | on);
14517| } __except(EXCEPTION_EXECUTE_HANDLER) {
14518|     Err = GetExceptionCode();
14519|     printf("MoveFile Exception %08x\n",Err);
14520| }
14521| return Err;
14522| }
14523|
14524| ULONG GetVolumeBitmapSize( HANDLE VolumeHandle )
14525| {
14526|     ULONG Err;
14527|     STARTING_VCN_INPUT_BUFFER SVIB={0};
14528|     VOLUME_BITMAP_BUFFER *VB;
14529|     ULONG VBSize =
        | (sizeof(VOLUME_BITMAP_BUFFER)-1)+4096;
14530|
14531|     VB = malloc( VBSize );
14532|     if(!VB) {
14533|         printf("Out of memory for VB\n");
14534|         return 0;
14535|     }
14536|     Err = FS_GetVolumeBitmap( VolumeHandle, &SVIB, VB,
        | VBSize );
14537|     if (Err!=STATUS_SUCCESS)
14538|         SetLastError(Err);
14539|
14540|     VBSize = (VB->BitmapSize.LowPart /
        | 8)+(sizeof(VOLUME_BITMAP_BUFFER)-1);
14541|     free(VB);
14542|     return VBSize;
14543| }
14544|
14545| ULONG UpdateVolumeBitmap( HANDLE VolumeHandle,
        | VOLUME_BITMAP_BUFFER *VB, ULONG VBSize )
14546| {
14547|     ULONG Err;
14548|     STARTING_VCN_INPUT_BUFFER SVIB={0};
14549|
14550|     Err = FS_GetVolumeBitmap( VolumeHandle, &SVIB, VB,
        | VBSize );
14551|     return Err;
14552| }
14553|
14554|
14555|
14556| File Listing: DEFRAG.h
14557|
14558| ULONG FS_GetNTFSFileRecord( HANDLE Handle,
        | NTFS_FILE_RECORD_INPUT_BUFFER *NFRI,

```

```

    | NTFS_FILE_RECORD_OUTPUT_BUFFER *NFRO, ULONG NFROSize );
14559| ULONG FS_GetNTFSVolumeData( HANDLE Handle,
    | NTFS_VOLUME_DATA_BUFFER *NVD );
14560| ULONG FS_GetVolumeBitmap( HANDLE Handle,
    | STARTING_VCN_INPUT_BUFFER *SVIB, VOLUME_BITMAP_BUFFER
    | *VB, ULONG VBSize );
14561| ULONG FS_GetRetrievalPointers( HANDLE Handle,
    | STARTING_VCN_INPUT_BUFFER *SVIB,
    | RETRIEVAL_POINTERS_BUFFER *RP, ULONG RPSize );
14562| ULONG FS_MoveFile( HANDLE Handle, MOVE_FILE_DATA *MF );
14563| ULONG GetVolumeBitmapSize( HANDLE VolumeHandle );
14564| ULONG UpdateVolumeBitmap( HANDLE VolumeHandle,
    | VOLUME_BITMAP_BUFFER *VB, ULONG VBSize );
14565| HANDLE OpenVolumeForDefrag( char Volume );
14566|
14567|
14568|
14569| File Listing: dlog.h
14570|
14571| #ifdef _DEBUG
14572|     extern ULONG DebugMode;
14573|     extern void PSM_LogDebugInfo ( const TCHAR *fmt,
    | ... );
14574|     #define DLOG(x) PSM_LogDebugInfo x
14575| #else
14576|     #define DLOG(x)
14577| #endif /*_DEBUG*/
14578|
14579| /*--- end of file dlog.h ---*/
14580|
14581|
14582|
14583| File Listing: dump.c
14584|
14585| #include <stdio.h>
14586| #include <stdlib.h>
14587| #include <string.h>
14588| #include <windows.h>
14589| #include <windowsx.h>
14590| #include <winioctl.h>
14591| #include <tchar.h>
14592| #include <conio.h>
14593|
14594| #include "mytypes.h"
14595| #include "dasd.h"
14596| #include "fs.h"
14597| #include "ntfs.h"
14598| #include "dump.h"
14599| #include "safemem.h"
14600| #include "dlog.h"

```

```

14601|
14602| //void DLOG(( const TCHAR *fmt,...);
14603|
14604| /*
14605|     Dumps all the runs of a file (what clusters the
        | file uses)
14606| */
14607| void DumpRun ( PVOID DataRun, ULONG NumberOfBytes )
14608| {
14609|     ULARGE_INTEGER Cluster={0};
14610|     ULARGE_INTEGER Length={0};
14611|     ULONG Offset=0;
14612|     ULONG Sparse=0;
14613|
14614|     | while(GetRun(DataRun,&Offset,&Cluster,&Length,&Sparse))
        | {
14615|         if(!Sparse) {
14616|             DLOG((TEXT("Cluster = %12I64d Length =
        | %12I64d\n"),Cluster,Length));
14617|         } else {
14618|             DLOG((TEXT("Compressed Length =
        | %12I64d\n"),Length));
14619|         }
14620|         if(Offset>=NumberOfBytes) {
14621|             DLOG((TEXT("Out of range!\n")));
14622|         }
14623|     }
14624| }
14625|
14626| void DumpBytes( PVOID Bytes, ULONG Length )
14627| {
14628|     ULONG i,j;
14629|     for(i=0;i<Length;i+=16) {
14630|         DLOG((TEXT("%3x: "),i));
14631|         for(j=i;j<(i+16<Length ? i+16 : Length-i);j++)
            | {
14632|             DLOG((TEXT("%02x "),((BYTE*)Bytes)[j]));
14633|         }
14634|         for(j=i;j<(i+16<Length ? i+16 : Length-i);j++)
            | {
14635|             DLOG((TEXT("%hc"),((BYTE*)Bytes)[j] > 31 ?
            | ((BYTE*)Bytes)[j] : '.'));
14636|         }
14637|         DLOG((TEXT("\n")));
14638|     }
14639| }
14640|
14641| void DumpBitMap( PVOID BitMap, ULONG Length )
14642| {

```

```

14643| BYTE *Bits=(BYTE*)BitMap;
14644| ULONG i,j;
14645|
14646| for(j=0;j<Length;j+=4) {
14647|     for(i=0;i<4;i++) {
14648|         DLOG((TEXT("%c%c%c%c%c%c%c%c%c "),
14649|             Bits[j+i] & 0x80 ? TEXT('1') :
14650|             | TEXT('0'),
14651|             Bits[j+i] & 0x40 ? TEXT('1') :
14652|             | TEXT('0'),
14653|             Bits[j+i] & 0x20 ? TEXT('1') :
14654|             | TEXT('0'),
14655|             Bits[j+i] & 0x10 ? TEXT('1') :
14656|             | TEXT('0'),
14657|             Bits[j+i] & 0x08 ? TEXT('1') :
14658|             | TEXT('0'),
14659|             Bits[j+i] & 0x04 ? TEXT('1') :
14660|             | TEXT('0'),
14661|             Bits[j+i] & 0x02 ? TEXT('1') :
14662|             | TEXT('0'),
14663|             Bits[j+i] & 0x01 ? TEXT('1') :
14664|             | TEXT('0')
14665|         ));
14666|     }
14667|     DLOG((TEXT(" ")));
14668|     for(i=0;i<4;i++) {
14669|         DLOG((TEXT("%02x "),Bits[j+i]));
14670|     }
14671|     for(i=0;i<4;i++) {
14672|         if(Bits[j+i]>31)
14673|             DLOG((TEXT("%c"),Bits[j+i]));
14674|         else
14675|             DLOG((TEXT(".")));
14676|     }
14677|     DLOG((TEXT("\n")));
14678| }
14679| }
14680|
14681| TCHAR TempString[128];
14682|
14683| TCHAR *GetDosAttributes( ULONG Attr )
14684| {
14685|     _tcsncpy(TempString,TEXT("....."));
14686|     if(Attr & DOS_READONLY)     TempString[0] =
14687|         | 'R';
14688|     if(Attr & DOS_HIDDEN)       TempString[1] =
14689|         | 'H';
14690|     if(Attr & DOS_SYSTEM)       TempString[2] =
14691|         | 'S';
14692|     if(Attr & DOS_ARCHIVE)     TempString[3] =

```

```

    | 'A';
14682|   if(Attr & DOS_COMPRESSED)   TempString[4] =
    | 'C';
14683|   if(Attr & FILENAME_DIRECTORY) TempString[5] =
    | 'D';
14684|   return TempString;
14685| }
14686|
14687| TCHAR *GetFileNameType( ULONG Type )
14688| {
14689|   switch(Type) {
14690|     case FILENAME_POSIX      : return
    | TEXT("Posix");
14691|     case FILENAME_UNICODE    : return
    | TEXT("Unicode");
14692|     case FILENAME_DOS        : return
    | TEXT("Dos");
14693|     case FILENAME_UNICODE_DOS : return
    | TEXT("Unicode Dos");
14694|     default:
14695|       return TEXT("Unknown File Type");
14696|   }
14697| }
14698|
14699| TCHAR *GetAttributeName( ULONG Type )
14700| {
14701|   switch(Type) {
14702|     case STANDARD_INFORMATION_ATTR : return
    | TEXT("Standard Information");
14703|     case ATTRIBUTE_LIST_ATTR       : return
    | TEXT("Attribute List");
14704|     case FILENAME_ATTR             : return
    | TEXT("Filename");
14705|     case VOLUME_VERSION_ATTR       : return
    | TEXT("Volume Version");
14706|     case SECURITY_DESCRIPTOR_ATTR   : return
    | TEXT("Security Descriptor");
14707|     case VOLUME_NAME_ATTR          : return
    | TEXT("Volume Name");
14708|     case VOLUME_INFO_ATTR          : return
    | TEXT("Volume Info");
14709|     case DATA_ATTR                : return
    | TEXT("Data");
14710|     case INDEX_ROOT_ATTR           : return
    | TEXT("Index Root");
14711|     case INDEX_ALLOCATION_ATTR      : return
    | TEXT("Index Allocation");
14712|     case BITMAP_ATTR               : return
    | TEXT("Bitmap");
14713|     case SYMLINK_ATTR              : return

```

```

    | TEXT("Symbolic Link");
14714|     case HPFS_EA_INFO_ATTR      : return
    | TEXT("HPFS EA Info");
14715|     case HPFS_EA_ATTR          : return
    | TEXT("HPFS EA");
14716|     default :
14717|         return TEXT("Unknown Attribute Type");
14718| }
14719| }
14720|
14721| /*
14722|     Time is the number of 100ns increments since Jan 1,
    | 1601. (Universal coordinated time)
14723| */
14724| TCHAR *GetTime( ULARGE_INTEGER Number )
14725| {
14726|     TCHAR *Days[7] = { TEXT("Sunday"),
14727|                        TEXT("Monday"),
14728|                        TEXT("Tuesday"),
14729|                        TEXT("Wednesday"),
14730|                        TEXT("Thursday"),
14731|                        TEXT("Friday"),
14732|                        TEXT("Saturday")};
14733|     TCHAR *Months[12] = { TEXT("January"),
14734|                           TEXT("February"),
14735|                           TEXT("March"),
14736|                           TEXT("April"),
14737|                           TEXT("May"),
14738|                           TEXT("June"),
14739|                           TEXT("July"),
14740|                           TEXT("August"),
14741|                           TEXT("September"),
14742|                           TEXT("October"),
14743|                           TEXT("November"),
14744|                           TEXT("December")};
14745|
14746|     FILETIME *FT=(FILETIME*)&Number;
14747|     SYSTEMTIME ST;
14748|
14749|     FileTimeToSystemTime( FT, &ST );
14750|
14751|     _sprintf(TempString,TEXT("%s %s %02d, %04d
    | %02d:%02d:%02d:%02d"),
14752|             Days[ST.wDayOfWeek],
14753|             Months[ST.wMonth-1],
14754|             ST.wDay,
14755|             ST.wYear,
14756|             ST.wHour,
14757|             ST.wMinute,
14758|             ST.wSecond,

```



```

14759|     ST.wMilliseconds
14760| );
14761| return TempString;
14762| }
14763|
14764| void DumpMft( pVolumeInfo VolumeInfo, PMFT Mft )
14765| {
14766|     PATTRIBUTE Attribute;
14767|     ULONG Where;
14768|
14769|     if(Mft->Signature == 'ELIF') {
14770|         DLOG((TEXT("Unknown      :
14771|         | %016l64x\n"),Mft->Unknown));
14772|         DLOG((TEXT("SequenceNumber  :
14773|         | %04x\n"),Mft->SequenceNumber));
14774|         DLOG((TEXT("HardLinkCount   :
14775|         | %04x\n"),Mft->HardLinkCount));
14776|         DLOG((TEXT("Flags          :
14777|         | %04x\n"),Mft->Flags));
14778|         DLOG((TEXT("LengthInUse     :
14779|         | %08x\n"),Mft->LengthInUse));
14780|         DLOG((TEXT("Allocated       :
14781|         | %08x\n"),Mft->Allocated));
14782|         DLOG((TEXT("MftRecord      :
14783|         | %016l64x\n"),Mft->MftRecord));
14784|         DLOG((TEXT("MaxAttributeNumber :
14785|         | %04x\n"),Mft->MaxAttributeNumber));
14786|
14787|         Attribute =
14788|         | (PATTRIBUTE)&(((char*)Mft)[Mft->OffsetToAttributes]);
14789|
14790|         Where = 0;
14791|         while(Attribute->Type!=0xffffffff) {
14792|
14793|             | DLOG((TEXT("-----
14794|             | -----\n")));
14795|             DLOG((TEXT("Type          : %08x
14796|             | (%s)\n"),Attribute->Type,GetAttributeName(Attribute->Typ
14797|             | e)));
14798|             DLOG((TEXT("Length        :
14799|             | %08x\n"),Attribute->Length));
14800|             DLOG((TEXT("Not Resident  :
14801|             | %08x\n"),Attribute->NotResident));
14802|             DLOG((TEXT("Name Length   :
14803|             | %08x\n"),Attribute->NameLength));
14804|             DLOG((TEXT("Offset        :
14805|             | %08x\n"),Attribute->Offset));
14806|             DLOG((TEXT("Compressed    :

```

```

    | %08x\n"),Attribute->Compressed));
14792|      DLOG((TEXT("AttributeID :
    | %08x\n"),Attribute->AttributeID));
14793|      if(Attribute->NotResident) {
14794|          DLOG((TEXT("StartingVCN :
    | %12l64d\n"),Attribute->NonResidentData.StartingVCN));
14795|          DLOG((TEXT("LastVCN :
    | %12l64d\n"),Attribute->NonResidentData.LastVCN));
14796|          DLOG((TEXT("Offset :
    | %08x\n"),Attribute->NonResidentData.Offset));
14797|          DLOG((TEXT("CompressionEngine :
    | %04x\n"),Attribute->NonResidentData.CompressionEngine));
14798|          DLOG((TEXT("AllocatedDiskSpace :
    | %12l64d\n"),Attribute->NonResidentData.AllocatedDiskSpace
    | e));
14799|          DLOG((TEXT("AttributeSize :
    | %12l64d\n"),Attribute->NonResidentData.AttributeSize));
14800|          DLOG((TEXT("LenOfInitializedData :
    | %12l64d\n"),Attribute->NonResidentData.LengthOfInitializ
    | edData));
14801|          if(Attribute->Compressed)
14802|              DLOG((TEXT("CompressedSize :
    | %12l64d\n"),Attribute->NonResidentData.CompressedSize));
14803|      } else {
14804|          DLOG((TEXT("DataLength :
    | %08x\n"),Attribute->ResidentData.DataLength));
14805|          DLOG((TEXT("Offset :
    | %08x\n"),Attribute->ResidentData.Offset));
14806|          DLOG((TEXT("AttributeIsIndexed :
    | %08x\n"),Attribute->ResidentData.AttributeIsIndexed));
14807|      }
14808|
14809|      if(Attribute->NameLength) {
14810|          DLOG((TEXT("Name Len=%d,
    | '%-*.s'\n"),Attribute->NameLength,Attribute->NameLength
    | ,Attribute->NameLength,((char*)&(((CHAR*)Attribute)[Attr
    | ibute->Offset]))));
14811|      }
14812|
14813|
14814|      switch(Attribute->Type) {
14815|          case STANDARD_INFORMATION_ATTR: {
14816|              PSTANDARD_INFORMATION SI =
    | NTFS_GetDataPointer(Attribute);
14817|              DLOG((TEXT("Standard information
    | read\n")));
14818|              DLOG((TEXT(" FileCreationTime
    | : %s\n"),GetTime(SI->FileCreationTime)));
14819|              DLOG((TEXT(" LastModifiedTime
    | : %s\n"),GetTime(SI->LastModifiedTime)));

```

```

14820|          DLOG((TEXT("  LastModifiedTimeMft
| : %s\n"),GetTime(SI->LastModifiedTimeForMFT)));
14821|          DLOG((TEXT("  LastAccessTime
| : %s\n"),GetTime(SI->LastAccessTime)));
14822|          DLOG((TEXT("  DOS Attributes
| : %08x
| (%s)\n"),SI->DosAttributes,GetDosAttributes(SI->DosAttri
| butes)));
14823|          break;
14824|      }
14825|      case ATTRIBUTE_LIST_ATTR: {
14826|          PATTRIBUTE_LIST AL =
| NTFS_GetDataPointer(Attribute);
14827|          DLOG((TEXT("Attribute list
| read\n")));
14828|          break;
14829|      }
14830|      case FILENAME_ATTR: {
14831|          PFILENAME FN =
| NTFS_GetDataPointer(Attribute);
14832|          DLOG((TEXT("FileName read (type=%d
| (%s), Len=%d)
| %-*. *s\n"),FN->FileNameType,GetFileNameType(FN->FileName
| Type),FN->FileNameLength,FN->FileNameLength,FN->FileName
| Length,FN->FileName)));
14833|          DLOG((TEXT("  Sequence Number
| : %04x\n"),FN->SequenceNumber));
14834|          DLOG((TEXT("  Directory Mft
| : %016l64x
| (%12l64d)\n"),FN->DirectoryMft,FN->DirectoryMft));
14835|          DLOG((TEXT("  FileCreationTime
| : %s\n"),GetTime(FN->FileCreationTime)));
14836|          DLOG((TEXT("  LastModifiedTime
| : %s\n"),GetTime(FN->LastModifiedTime)));
14837|          DLOG((TEXT("  LastModifiedTimeMft
| : %s\n"),GetTime(FN->LastModifiedTimeForMFT)));
14838|          DLOG((TEXT("  LastAccessTime
| : %s\n"),GetTime(FN->LastAccessTime)));
14839|          DLOG((TEXT("  FileSize
| : %12l64d\n"),FN->FileSize));
14840|          DLOG((TEXT("  AttributeSize
| : %12l64d\n"),FN->AttributeSize));
14841|          DLOG((TEXT("  Flags
| : %08x
| (%s)\n"),FN->Flags,GetDosAttributes(FN->Flags)));
14842|          break;
14843|      }
14844|      case VOLUME_VERSION_ATTR: {
14845|          PBYTE_ARRAY BA =
| NTFS_GetDataPointer(Attribute);

```

```

14846|           DLOG((TEXT("Volume Version
| read\n"))));
14847|           break;
14848|       }
14849|       case SECURITY_DESCRIPTOR_ATTR: {
14850|           PBYTE_ARRAY BA =
| NTFS_GetDataPointer(Attribute);
14851|           DLOG((TEXT("Security Descriptor
| read\n"))));
14852|           break;
14853|       }
14854|       case VOLUME_NAME_ATTR: {
14855|           WCHAR *VolName =
| NTFS_GetDataPointer(Attribute);
14856|           DLOG((TEXT("Volume Name read
| (Len=%d,
| '%-*.*s'\n"),Attribute->ResidentData.DataLength,Attribu
| te->ResidentData.DataLength /
| 2,Attribute->ResidentData.DataLength / 2,VolName)));
14857|           break;
14858|       }
14859|       case VOLUME_INFO_ATTR: {
14860|           PVOLUME_INFORMATION VI =
| NTFS_GetDataPointer(Attribute);
14861|           DLOG((TEXT("Volume Info read
| (Chkdsk /f=%s)\n"),VI->Chkdsk ? TEXT("Yes") :
| TEXT("No"))));
14862|           | DumpBytes(VI,Attribute->Length-Attribute->Offset);
14863|           break;
14864|       }
14865|       case DATA_ATTR: {
14866|           PBYTE_ARRAY BA =
| NTFS_GetDataPointer(Attribute);
14867|           DLOG((TEXT("Data read\n"))));
14868|           break;
14869|       }
14870|       case INDEX_ROOT_ATTR: {
14871|           PINDEX_ROOT IR =
| NTFS_GetDataPointer(Attribute);
14872|           DLOG((TEXT("Index Root read\n"))));
14873|           break;
14874|       }
14875|       case INDEX_ALLOCATION_ATTR: {
14876|           PINDEX_ALLOCATION IA =
| NTFS_GetDataPointer(Attribute);
14877|           DLOG((TEXT("Index Allocation
| read\n"))));
14878|           break;
14879|       }

```

```

14880|         case BITMAP_ATTR: {
14881|             PBYTE_ARRAY BA =
14882|             | NTFS_GetDataPointer(Attribute);
14883|             DLOG((TEXT("Bitmap read\n")));
14884|             break;
14885|         }
14886|         case SYMLINK_ATTR: {
14887|             PBYTE_ARRAY BA =
14888|             | NTFS_GetDataPointer(Attribute);
14889|             DLOG((TEXT("SymLink read\n")));
14890|             break;
14891|         }
14892|         case HPFS_EA_INFO_ATTR: {
14893|             PBYTE_ARRAY BA =
14894|             | NTFS_GetDataPointer(Attribute);
14895|             DLOG((TEXT("EA Info read\n")));
14896|             break;
14897|         }
14898|         case HPFS_EA_ATTR: {
14899|             PBYTE_ARRAY BA =
14900|             | NTFS_GetDataPointer(Attribute);
14901|             DLOG((TEXT("EA Attributes
14902|             | read\n")));
14903|             break;
14904|         }
14905|         default: {
14906|             PBYTE_ARRAY BA =
14907|             | NTFS_GetDataPointer(Attribute);
14908|             DLOG((TEXT("Unknown attribute
14909|             | encountered\n")));
14910|             break;
14911|         }
14912|     }
14913|     // dump cluster numbers where attribute
14914|     | resides
14915|     if(Attribute->NotResident) {
14916|         PBYTE_ARRAY BA =
14917|         | NTFS_GetDataPointer(Attribute);
14918|         DumpRun(BA,Attribute->Length-Attribute->Offset);
14919|     }
14920|     Where += Attribute->Length;
14921|     Attribute =
14922|     | (PATTRIBUTE)&(((char*)Mft)[Mft->OffsetToAttributes+Where
14923|     | ]);
14924| }
14925| }
14926| }

```

```

    | DumpBytes(Mft,VolumInfo->NtfsBootSector->MFTRecordSize*
    | VolumInfo->NtfsBootSector->BytesPerSector);
14918| } else {
14919|     DLOG((TEXT("Mft not found\n")));
14920| }
14921| }
14922|
14923| void DumpNtfsBootSector( PNTFS_BOOT_SECTOR
    | NtfsBootSector )
14924| {
14925|     DLOG((TEXT("BytesPerSector :
    | %d\n"),NtfsBootSector->BytesPerSector));
14926|     DLOG((TEXT("SectorsPerCluster :
    | %d\n"),NtfsBootSector->SectorsPerCluster));
14927|     DLOG((TEXT("MediaID :
    | %02hx\n"),NtfsBootSector->MediaId));
14928|     DLOG((TEXT("SectorsPerTrack :
    | %d\n"),NtfsBootSector->SectorsPerTrack));
14929|     DLOG((TEXT("Heads :
    | %d\n"),NtfsBootSector->Heads));
14930|     DLOG((TEXT("Drive :
    | %02hx\n"),NtfsBootSector->Drive));
14931|     DLOG((TEXT("DirtyVolume :
    | %d\n"),NtfsBootSector->DirtyVolume));
14932|     DLOG((TEXT("NumberOfSectors :
    | %12l64d\n"),NtfsBootSector->NumberOfSectors));
14933|     DLOG((TEXT("MftCluster :
    | %12l64d\n"),NtfsBootSector->MftCluster));
14934|     DLOG((TEXT("Mft2Cluster :
    | %12l64d\n"),NtfsBootSector->Mft2Cluster));
14935|     DLOG((TEXT("MFTRecordSize :
    | %d\n"),NtfsBootSector->MFTRecordSize));
14936|     DLOG((TEXT("IndexBufferSize :
    | %d\n"),NtfsBootSector->IndexBufferSize));
14937|     DLOG((TEXT("SerialNumber :
    | %l64x\n"),NtfsBootSector->SerialNumber));
14938| }
14939|
14940| void DumpNtfsStructures()
14941| {
14942|     // Verify everything starts where it should
14943|     DLOG((TEXT("Type =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,Type)));
14944|     DLOG((TEXT("Length =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,Length)));
14945|     DLOG((TEXT("NotResident =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NotResident)));
14946|     DLOG((TEXT("NameLength =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NameLength)));
14947|     DLOG((TEXT("Offset =

```

```

    | %x\n"),FIELD_OFFSET(ATTRIBUTE,Offset)));
14948|  DLOG((TEXT("Compressed          =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,Compressed)));
14949|  DLOG((TEXT("AttributeId          =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,AttributeId)));
14950|
14951|  DLOG((TEXT("Resident Data\n")));
14952|  DLOG((TEXT("  DataLength          =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,ResidentData.DataLength)))
    | ;
14953|  DLOG((TEXT("  Offset              =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,ResidentData.Offset)));
14954|  DLOG((TEXT("  AttributeIsIndexed    =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,ResidentData.AttributeIsIn
    | dexed)));
14955|  DLOG((TEXT("  Data                  =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,ResidentData.AttributeIsIn
    | dexed)+2));
14956|
14957|  DLOG((TEXT("NonResident Data\n")));
14958|  DLOG((TEXT("  StartingVCN          =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.StartingVC
    | N)));
14959|  DLOG((TEXT("  LastVCN              =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.LastVCN)))
    | ;
14960|  DLOG((TEXT("  Offset              =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.Offset)));
14961|  DLOG((TEXT("  CompressionEngine      =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.Compressio
    | nEngine)));
14962|  DLOG((TEXT("  AllocatedDiskSpace    =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.AllocatedD
    | iskSpace)));
14963|  DLOG((TEXT("  AttributeSize         =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.AttributeS
    | ize)));
14964|  DLOG((TEXT("  LengthOfInitializedData =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.LengthOfIn
    | itializedData)));
14965|  DLOG((TEXT("  CompressedSize        =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.Compressed
    | Size)));
14966|  DLOG((TEXT("  Data                  =
    | %x\n"),FIELD_OFFSET(ATTRIBUTE,NonResidentData.Compressed
    | Size)+8));
14967|
14968|  DLOG((TEXT("INDEX_ENTRY\n")));
14969|  DLOG((TEXT("  EntrySize            =
    | %x\n"),FIELD_OFFSET(INDEX_ENTRY,EntrySize)));

```

```

14970|  DLOG((TEXT("  Flags          =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,Flags)));
14971|  DLOG((TEXT("  FileCreationTime    =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,FileCreationTime)));
14972|  DLOG((TEXT("  LastModifiedTime     =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,LastModifiedTime)));
14973|  DLOG((TEXT("  LastModifiedTimeForMft =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,LastModifiedTimeForMFT))
      | );
14974|  DLOG((TEXT("  LastAccessTime       =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,LastAccessTime)));
14975|  DLOG((TEXT("  AllocatedAttrSize    =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,AllocatedAttributeSize))
      | );
14976|  DLOG((TEXT("  AttributeSize        =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,AttributeSize)));
14977|  DLOG((TEXT("  FileNameLength       =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,FileNameLength)));
14978|  DLOG((TEXT("  FileNameType         =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,FileNameType)));
14979|  DLOG((TEXT("  FileName             =
      | %x\n"),FIELD_OFFSET(INDEX_ENTRY,FileName)));
14980|
14981|  DLOG((TEXT("sizeof(NTFS_BOOT_SECTOR) =
      | %d\n"),sizeof(NTFS_BOOT_SECTOR)));
14982| }
14983|
14984| void CreateMultipleStreamFile()
14985| {
14986|  HANDLE F;
14987|  ULONG NumWritten;
14988|  char *Buffer;
14989|  int i;
14990|
14991|  Buffer=SAFE_Alloc(512);
14992|  // create unnamed stream
14993|  F = CreateFile(
14994|      TEXT("MultiStreamFile"), // pointer to name
      | of the file
14995|      GENERIC_WRITE, // access (read-write) mode
14996|      FILE_SHARE_READ | FILE_SHARE_WRITE, // share
      | mode
14997|      NULL, // pointer to security descriptor
14998|      CREATE_ALWAYS, // how to create
14999|      FILE_ATTRIBUTE_NORMAL, // file attributes
15000|      0 // handle to file with attributes to copy
15001|  );
15002|  if(F!=INVALID_HANDLE_VALUE) {
15003|      for(i=0;i<512;i+=16) {
15004|          strncpy(Buffer+i,"Unnamed stream ",16);

```



```

15005|     }
15006|     WriteFile(F,Buffer,512,&NumWritten,NULL);
15007|
15008|     CloseHandle(F);
15009| }
15010|
15011| // create "backup" stream
15012| F = CreateFile(
15013|     TEXT("MultiStreamFile:backup"), // pointer to
    | name of the file
15014|     GENERIC_WRITE, // access (read-write) mode
15015|     FILE_SHARE_READ | FILE_SHARE_WRITE, // share
    | mode
15016|     NULL, // pointer to security descriptor
15017|     CREATE_ALWAYS, // how to create
15018|     FILE_ATTRIBUTE_NORMAL, // file attributes
15019|     0 // handle to file with attributes to copy
15020| );
15021| if(F!=INVALID_HANDLE_VALUE) {
15022|     for(i=0;i<512;i+=16) {
15023|         strncpy(Buffer+i,"Backup stream ",16);
15024|     }
15025|     WriteFile(F,Buffer,512,&NumWritten,NULL);
15026|
15027|     CloseHandle(F);
15028| }
15029|
15030| // create "resource" stream
15031| F = CreateFile(
15032|     TEXT("MultiStreamFile:resource"), // pointer
    | to name of the file
15033|     GENERIC_WRITE, // access (read-write) mode
15034|     FILE_SHARE_READ | FILE_SHARE_WRITE, // share
    | mode
15035|     NULL, // pointer to security descriptor
15036|     CREATE_ALWAYS, // how to create
15037|     FILE_ATTRIBUTE_NORMAL, // file attributes
15038|     0 // handle to file with attributes to copy
15039| );
15040| if(F!=INVALID_HANDLE_VALUE) {
15041|     for(i=0;i<512;i+=16) {
15042|         strncpy(Buffer+i,"Resource Stream ",16);
15043|     }
15044|     WriteFile(F,Buffer,512,&NumWritten,NULL);
15045|
15046|     CloseHandle(F);
15047| }
15048| SAFE_Free(Buffer);
15049| }
15050|

```

```

15051| void CopyAndCompareFile ( pVolumeInfo VolumeInfo )
15052| {
15053|     PNTFS_File File;
15054|     char
        | *Buffer=SAFE_Alloc(NTFS_ClusterSizeInBytes(VolumeInfo));
15055|     char
        | *Buffer2=SAFE_Alloc(NTFS_ClusterSizeInBytes(VolumeInfo))
        | ;
15056|
15057|     if(Buffer) {
15058|         HANDLE F;
15059|         ULONG Read;
15060|
15061|         F = CreateFile(
15062|             TEXT("d:\\winnt\\inf\\java.pnf"), //
        | pointer to name of the file
15063|             GENERIC_READ, // access (read-write) mode
15064|             FILE_SHARE_READ | FILE_SHARE_WRITE, //
        | share mode
15065|             NULL, // pointer to security descriptor
15066|             OPEN_EXISTING, // how to create
15067|             FILE_ATTRIBUTE_NORMAL, // file attributes
15068|             0 // handle to file with attributes to
        | copy
15069|         );
15070|         if(F==INVALID_HANDLE_VALUE)
15071|             abort();
15072|
15073|         File = NTFS_OpenFileByNumber( VolumeInfo, 61,
        | DATA_ATTR, NULL );
15074|         if(File) {
15075|             ULONGLONG FileSize =
        | NTFS_GetFileSize(File);
15076|             ULONGLONG i=0;
15077|             ULONGLONG LCN;
15078|             PATTRIBUTE
        | Attribute=NTFS_GetAttributeByType( File->VolumeInfo,
        | File->Mft, DATA_ATTR );
15079|             DumpMft(VolumeInfo,File->Mft);
15080|             if(Attribute)
15081|
        | DumpBytes(NTFS_GetDataPointer(Attribute),Attribute->Leng
        | th-Attribute->NonResidentData.Offset);
15082|
15083|
        | for(i=0;i<FileSize;i+=NTFS_ClusterSizeInBytes(VolumeInfo
        | )) {
15084|                 LCN = NTFS_VCNTToLCN( File,
        | i/NTFS_ClusterSizeInBytes(VolumeInfo));
15085|

```

```

    | memset(Buffer,0,NTFS_ClusterSizeInBytes(VolumeInfo));
15086|
    | memset(Buffer2,0,NTFS_ClusterSizeInBytes(VolumeInfo));
15087|     NTFS_ReadFile( File, i,
    | NTFS_ClusterSizeInBytes(VolumeInfo), Buffer );
15088|     SetFilePointer(
15089|         F, // handle of file
15090|         (ULONG)i, // number of bytes to
    | move file pointer
15091|         NULL, // address of high-order
    | word of distance to move
15092|         FILE_BEGIN // how to move
15093|     );
15094|     ReadFile( F, // handle of file to
    | read
15095|         Buffer2, // address of buffer
    | that receives data
15096|
    | NTFS_ClusterSizeInBytes(VolumeInfo), // number of
    | bytes to read
15097|         &Read, // address of number of
    | bytes read
15098|         NULL// address of structure for
    | data
15099|     );
15100|     DLOG((TEXT("Offset=%12l64d,
    | VCN=%12l64d,
    | LCN=%12l64d\n"),i,i/NTFS_ClusterSizeInBytes(VolumeInfo),
    | LCN));
15101|
    | if(memcmp(Buffer,Buffer2,NTFS_ClusterSizeInBytes(VolumeI
    | nfo))!=0) {
15102|
    | DumpBytes(Buffer,NTFS_ClusterSizeInBytes(VolumeInfo));
15103|         DLOG((TEXT("Should be\n")));
15104|
    | DumpBytes(Buffer2,NTFS_ClusterSizeInBytes(VolumeInfo));
15105|         DLOG((TEXT("Data is
    | different!\n"))));
15106|     }
15107| }
15108|     NTFS_CloseFile(File);
15109| }
15110|     CloseHandle(F);
15111|
15112|     SAFE_Free(Buffer2);
15113|     SAFE_Free(Buffer);
15114| }
15115| }
15116|

```

```

15117| #if 0
15118| void DumpMftEntryNumber( ULONG Drive, ULONG Part,
    | ULONGLONG MftEntry )
15119| {
15120|     ULONG PartCount=0;
15121|     ULONGLONG Sector=0;
15122|     tDASDHandle *LockHandle=NULL;
15123|     tVolumeInfo *VolumeInfo=NULL;
15124|     ULONG Err=0;
15125|     PMFT Mft;
15126|
15127|     LockHandle = DASD_LockDevice( Drive );
15128|     if(LockHandle) {
15129|         Sector = CountPartitionTable( LockHandle, 0,
            | &PartCount, Part );
15130|         if(Sector!=(ULONGLONG)-1) {
15131|             VolumeInfo = NTFS_OpenVolume( LockHandle,
                | Sector );
15132|             if(VolumeInfo) {
15133|                 Mft =
                    | (PMFT)SAFE_Alloc(NTFS_MftByteSize(VolumeInfo));
15134|                 if(Mft) {
15135|                     Err = NTFS_ReadMftEntryNumber(
                        | VolumeInfo, MftEntry, Mft );
15136|                     DumpMft( VolumeInfo, Mft );
15137|                     SAFE_Free(Mft);
15138|                 } else {
15139|                     DLOG((TEXT("Out of memory\n")));
15140|                 }
15141|
15142|                 NTFS_CloseVolume( VolumeInfo );
15143|             } else {
15144|                 DLOG((TEXT("Error %08x opening
                    | volume\n"),GetLastError()));
15145|             }
15146|         } else {
15147|             DLOG((TEXT("Invalid partition\n")));
15148|         }
15149|         DASD_UnlockDevice(LockHandle);
15150|     } else {
15151|         DLOG((TEXT("Invalid drive\n")));
15152|     }
15153| }
15154| #endif
15155|
15156| void ListMftEntries( pVolumeInfo VolumeInfo )
15157| {
15158|     char
        | *Buffer=SAFE_Alloc(NTFS_ClusterSizeInBytes(VolumeInfo));
15159|     char

```

```

    | *Buffer2=SAFE_Alloc(NTFS_ClusterSizeInBytes(VolumeInfo))
    | ;
15160|   PNTFS_File File;
15161|   ULONG MftEntry;
15162|
15163|   if(Buffer) {
15164|       for(MftEntry=0;MftEntry<1000;MftEntry++) {
15165|           DLOG((TEXT("----- %4d
    | -----\n"),MftEntry));
15166|           File = NTFS_OpenFileByNumber( VolumeInfo,
    | MftEntry, DATA_ATTR, NULL );
15167|
15168|           if(File) {
15169|               PATTRIBUTE
    | Attribute=NTFS_GetAttributeByType( File->VolumeInfo,
    | File->Mft, FILENAME_ATTR );
15170|               if(Attribute) {
15171|                   PFILENAME FN =
    | NTFS_GetDataPointer(Attribute);
15172|                   ULONG NameLength =
    | FN->FileNameLength;
15173|
15174|                   DLOG((TEXT("FileName='%-*.*s'\n"),NameLength,NameLength,
    | FN->FileName));
15175|               } else {
15176|                   DLOG((TEXT("No filename attribute
    | found\n")));
15177|               }
15178|               NTFS_CloseFile(File);
15179|           }
15180|       }
15181|       SAFE_Free(Buffer2);
15182|       SAFE_Free(Buffer);
15183|   }
15184| }
15185|
15186| void DumpNTFSPartition( pVolumeInfo VolumeInfo )
15187| {
15188|   PMFT Mft;
15189|   PNTFS_File File;
15190|
15191|   /*
15192|   ULONG Err;
15193|   DumpNtfsStructures();
15194|   DLOG((TEXT("=====
    | NtfsBootSector=====\n")));
15195|   DumpNtfsBootSector( VolumeInfo->NtfsBootSector );
15196|   DLOG((TEXT("===== FILE_MFT
    | =====\n")));

```

```

15197|   DumpMft( VolumeInfo, VolumeInfo->Mft ); // FILE_MFT
15198| */
15199|   Mft =
      | (PMFT)SAFE_Alloc(NTFS_MftByteSize(VolumeInfo));
15200|   if(Mft) {
15201|
15202|       DLOG((TEXT("Number of clusters = %l64d, Number
      | of free clusters =
      | %l64d\n"),NTFS_GetVolumeSizeInClusters(VolumeInfo),
      | NTFS_GetNumFreeClusters(VolumeInfo)));
15203|
15204|       File = NTFS_OpenFileByName(VolumeInfo,
      | L"\\winnt\\system32\\calc.exe", DATA_ATTR, NULL );
15205|       if(File) {
15206|           NTFS_CloseFile(File);
15207|       } else {
15208|           DLOG((TEXT("Error %08x opening
      | file\n"),GetLastError()));
15209|       }
15210|
15211| /*
15212|   DLOG((TEXT("===== FILE_MFTMIRR
      | =====\n")));
15213|   Err = NTFS_ReadMftEntryNumber( VolumeInfo,
      | FILE_MFTMIRR, Mft ); DumpMft( VolumeInfo, Mft );
15214|   DLOG((TEXT("===== FILE_LOGFILE
      | =====\n")));
15215|   Err = NTFS_ReadMftEntryNumber( VolumeInfo,
      | FILE_LOGFILE, Mft ); DumpMft( VolumeInfo, Mft );
15216|   DLOG((TEXT("===== FILE_VOLUME
      | =====\n")));
15217|   Err = NTFS_ReadMftEntryNumber( VolumeInfo,
      | FILE_VOLUME, Mft ); DumpMft( VolumeInfo, Mft );
15218|   DLOG((TEXT("===== FILE_ATTRDEF
      | =====\n")));
15219|   Err = NTFS_ReadMftEntryNumber( VolumeInfo,
      | FILE_ATTRDEF, Mft ); DumpMft( VolumeInfo, Mft );
15220|   DLOG((TEXT("===== FILE_ROOT
      | =====\n")));
15221|   Err = NTFS_ReadMftEntryNumber( VolumeInfo,
      | FILE_ROOT, Mft ); DumpMft( VolumeInfo, Mft );
15222|   DLOG((TEXT("===== FILE_BITMAP
      | =====\n")));
15223|   Err = NTFS_ReadMftEntryNumber( VolumeInfo,
      | FILE_BITMAP, Mft ); DumpMft( VolumeInfo, Mft );
15224|   DLOG((TEXT("===== FILE_BOOT
      | =====\n")));
15225|   Err = NTFS_ReadMftEntryNumber( VolumeInfo,
      | FILE_BOOT, Mft ); DumpMft( VolumeInfo, Mft );
15226|   DLOG((TEXT("===== FILE_BADCLUS

```

```

| =====\n"));
15227|     Err = NTFS_ReadMftEntryNumber( VolumeInfo,
| FILE_BADCLUS, Mft ); DumpMft( VolumeInfo, Mft );
15228|     DLOG((TEXT("===== FILE_QUOTA
| =====\n")));
15229|     Err = NTFS_ReadMftEntryNumber( VolumeInfo,
| FILE_QUOTA, Mft ); DumpMft( VolumeInfo, Mft );
15230|     DLOG((TEXT("===== FILE_UPCASE
| =====\n")));
15231|     Err = NTFS_ReadMftEntryNumber( VolumeInfo,
| FILE_UPCASE, Mft ); DumpMft( VolumeInfo, Mft );
15232|     DLOG((TEXT("===== 17
| =====\n")));
15233|     Err = NTFS_ReadMftEntryNumber( VolumeInfo, 17,
| Mft ); DumpMft( VolumeInfo, Mft );
15234|     DLOG((TEXT("===== 18
| =====\n")));
15235|     Err = NTFS_ReadMftEntryNumber( VolumeInfo, 18,
| Mft ); DumpMft( VolumeInfo, Mft );
15236|     DLOG((TEXT("===== 21
| =====\n")));
15237|     Err = NTFS_ReadMftEntryNumber( VolumeInfo, 21,
| Mft ); DumpMft( VolumeInfo, Mft );
15238| */
15239|     SAFE_Free(Mft);
15240| } else {
15241|     DLOG((TEXT("Out of memory\n")));
15242| }
15243| }
15244|
15245| void DumpFATPartition( pVolumeInfo VolumeInfo )
15246| {
15247| }
15248|
15249|
15250|
15251| File Listing: dump.h
15252|
15253| // diagnostics functions
15254| void DumpNTFSPartition( pVolumeInfo VolumeInfo );
15255| void DumpNtfsBootSector( PNTFS_BOOT_SECTOR
| NtfsBootSector );
15256| void DumpMft( pVolumeInfo VolumeInfo, PMFT Mft );
15257| void DumpBitMap( PVOID BitMap, ULONG Length );
15258| void DumpRun ( PVOID DataRun, ULONG NumberOfBytes );
15259| void DumpBytes( PVOID Bytes, ULONG Length );
15260| void DumpMftEntryNumber( ULONG Drive, ULONG Part,
| ULONGLONG MftEntry );
15261| void CreateMultipleStreamFile();
15262|

```

```

15263| void DumpFATPartition( pVolumeInfo VolumeInfo );
15264|
15265| // dump help routines
15266| TCHAR *GetTime( ULARGE_INTEGER Number );
15267| TCHAR *GetAttributeName( ULONG Type );
15268| TCHAR *GetFileNameType( ULONG Type );
15269| TCHAR *GetDosAttributes( ULONG Attr );
15270|
15271|
15272|
15273| File Listing: fs.c
15274|
15275| #include <stdio.h>
15276| #include <stdlib.h>
15277| #include <string.h>
15278| #include <windows.h>
15279| #include <windowsx.h>
15280| #include <winioctl.h>
15281| #include <tchar.h>
15282| #include <conio.h>
15283|
15284| #include "mytypes.h"
15285| #include "dasd.h"
15286| #include "fs.h"
15287|
15288| TCHAR *GetPartitionType ( BYTE Type )
15289| {
15290|     switch (Type) {
15291|         case PARTITION_Empty      : return TEXT(" ");
15292|         case PARTITION_DOS12Bit   : return TEXT(" DOS
| 12 bit");
15293|         case PARTITION_DOS16Bit   : return TEXT(" DOS
| 16 bit");
15294|         case PARTITION_DOSExtended : return TEXT(" DOS
| Extended");
15295|         case PARTITION_DOSHuge    : return TEXT(" DOS
| Huge");
15296|         case PARTITION_NTFS       : return TEXT("
| NTFS");
15297|         case PARTITION_DRDOSHuge  : return TEXT(" Dr.
| DOS Huge");
15298|         case PARTITION_DRDOSExtended: return TEXT(" Dr.
| DOS Extended");
15299|         case PARTITION_DRDOSCompress: return TEXT(" Dr.
| DOS Compressed");
15300|         case PARTITION_ConCurDOS : return TEXT("
| Concurrent DOS");
15301|         case PARTITION_Netware286 : return TEXT("
| NetWare 286");
15302|         case PARTITION_Netware386 : return TEXT("

```



```

    | NetWare 386");
15303|     default:
15304|         return TEXT("Unknown");
15305|     }
15306| }
15307|
15308| void ListPartitionTable ( tDASDHandle *LockHandle,
    | ULONGLONG Sector, ULONG *PartCount )
15309| {
15310|     ULONG i;
15311|     tMBR MBR;
15312|     ULONG Err;
15313|
15314|     //_tprintf(TEXT("Reading sector %12l64d (Partition
    | Table)\n"),Sector);
15315|     Err = DASD_Read ( LockHandle, Sector, 1, &MBR );
15316|     for(i=0;i<4;i++) {
15317|         if(MBR.Part[i].SystemType ==
    | PARTITION_DOSExtended ) {
15318|             // recursively call ourselves
15319|             ListPartitionTable( LockHandle,
    | MBR.Part[i].StartSector+Sector, PartCount );
15320|         } else {
15321|             if(MBR.Part[i].SystemType!=PARTITION_Empty)
    | {
15322|                 _tprintf(TEXT("%02d: %02x %02x %02x
    | %02x %02x %02x %02x %02x %08x %08x %08x %s\n"),
15323|                     (*PartCount)++,
15324|                     MBR.Part[i].Bootable,
15325|                     MBR.Part[i].BeginHead,
15326|                     MBR.Part[i].BeginSector,
15327|                     MBR.Part[i].BeginCyl,
15328|                     MBR.Part[i].SystemType,
15329|                     MBR.Part[i].EndHead,
15330|                     MBR.Part[i].EndSector,
15331|                     MBR.Part[i].EndCyl,
15332|                     MBR.Part[i].StartSector,
15333|                     MBR.Part[i].NumberOfSectors,
15334|
    | MBR.Part[i].StartSector+(ULONG)Sector,
15335|
    | GetPartitionType(MBR.Part[i].SystemType)
15336|             );
15337|         }
15338|     }
15339| }
15340| }
15341|
15342| void ListPartitions( ULONG DriveNum )
15343| {

```

```

15344|  tDASDHandle *LockHandle;
15345|  ULONG PartCount=0;
15346|
15347|  _tprintf(TEXT("    Beginning Ending\n"));
15348|  _tprintf(TEXT("##: Bt Hd Sc Cy Ty Hd Sc Cy Start
    | Num    VolStart Type\n"));
15349|  LockHandle = DASD_LockDevice( DriveNum );
15350|  if(LockHandle) {
15351|      ListPartitionTable( LockHandle, 0, &PartCount
    | );
15352|
15353|      DASD_UnlockDevice( LockHandle );
15354|  } else {
15355|      _tprintf(TEXT("Unable to obtain lock to
    | device\n"));
15356|  }
15357| }
15358|
15359| ULONGLONG CountPartitionTable( tDASDHandle *LockHandle,
    | ULONGLONG Sector, ULONG *PartCount, ULONG LookingFor )
15360| {
15361|  ULONG i;
15362|  tMBR MBR;
15363|  ULONG Err;
15364|  ULONGLONG SectorStart;
15365|
15366|  //_tprintf(TEXT("Reading sector %12l64d (Partition
    | Table)\n"),Sector);
15367|  Err = DASD_Read ( LockHandle, Sector, 1, &MBR );
15368|  for(i=0;i<4;i++) {
15369|      if(MBR.Part[i].SystemType ==
    | PARTITION_DOSExtended ) {
15370|          // recursively call ourselves
15371|          SectorStart = CountPartitionTable(
    | LockHandle, MBR.Part[i].StartSector+Sector, PartCount,
    | LookingFor );
15372|          if(SectorStart!=(ULONGLONG)-1)
15373|              return SectorStart;
15374|      } else {
15375|          if ((*PartCount) == LookingFor)
15376|              return MBR.Part[i].StartSector+Sector;
15377|          (*PartCount)++;
15378|      }
15379|  }
15380|  // part not found
15381|  return (ULONGLONG)-1;
15382| }
15383|
15384|
15385| ULONG GetPartitionTableLow( tDASDHandle *LockHandle,

```

```

    | ULONGLONG Sector, ULONG Max, ULONG *PartCount,
    | tPartTable *Parts )
15386| {
15387|     ULONG i;
15388|     tMBR MBR;
15389|     ULONG Err;
15390|
15391|     //_tprintf(TEXT("Reading sector %12I64d (Partition
    | Table)\n"),Sector);
15392|     Err = DASD_Read ( LockHandle, Sector, 1, &MBR );
15393|     for(i=0;i<4;i++) {
15394|         if(MBR.Part[i].SystemType ==
    | PARTITION_DOSExtended ) {
15395|             // recursively call ourselves
15396|             GetPartitionTableLow( LockHandle,
    | MBR.Part[i].StartSector+Sector, Max, PartCount, Parts
    | );
15397|         } else {
15398|             if(MBR.Part[i].SystemType !=
    | PARTITION_Empty) {
15399|                 if(*PartCount<Max) {
15400|                     Parts[*PartCount].SystemType =
    | MBR.Part[i].SystemType;
15401|                     Parts[*PartCount].StartSector =
    | MBR.Part[i].StartSector+Sector;
15402|                     Parts[*PartCount].NumberOfSectors =
    | MBR.Part[i].NumberOfSectors;
15403|                     (*PartCount)++;
15404|                 } else {
15405|                     // past how much is allocated
15406|                     return (ULONG)-1;
15407|                 }
15408|             }
15409|         }
15410|     }
15411|     return *PartCount;
15412| }
15413|
15414| /*
15415|  Gets a list of partitions on the volume
15416|  returns the count of partitions found or -1 if Max
    | partitions have been reached
15417|  Parts is filled in
15418| */
15419| ULONG GetPartitionTable( tDASDHandle *LockHandle, ULONG
    | Max, tPartTable *Parts )
15420| {
15421|     ULONG PartitionCount=0;
15422|     return GetPartitionTableLow( LockHandle, 0, Max,
    | &PartitionCount, Parts );

```

```

15423| }
15424|
15425|
15426|
15427| File Listing: fs.h
15428|
15429| #include "fs_ondisk.h"
15430| #include "ntfs_ondisk.h"
15431|
15432| // for partition scanning
15433| typedef struct sPartTable {
15434|     BYTE      SystemType;
15435|     ULONGLONG  StartSector;
15436|     ULONGLONG  NumberOfSectors;
15437| } tPartTable, *pPartTable;
15438|
15439| TCHAR *GetPartitionType ( BYTE Type );
15440| void ListPartitionTable ( tDASDHandle *LockHandle,
    | ULONGLONG Sector, ULONG *PartCount );
15441| void ListPartitions( ULONG DriveNum );
15442| ULONGLONG CountPartitionTable( tDASDHandle *LockHandle,
    | ULONGLONG Sector, ULONG *PartCount, ULONG LookingFor );
15443| ULONG GetPartitionTableLow( tDASDHandle *LockHandle,
    | ULONGLONG Sector, ULONG Max, ULONG *PartCount,
    | tPartTable *Parts );
15444| ULONG GetPartitionTable( tDASDHandle *LockHandle, ULONG
    | Max, tPartTable *Parts );
15445|
15446|
15447|
15448| File Listing: fs_ondisk.h
15449|
15450| /* general partition stuff */
15451|
15452| #define PARTITION_Empty          0x00
15453| #define PARTITION_DOS12Bit      0x01 // 12 bit fat
    | < 10mb
15454| #define PARTITION_DOS16Bit      0x04 // 16 bit fat
    | < 32mb
15455| #define PARTITION_DOSExtended   0x05 // Extended
    | Dos Partition
15456| #define PARTITION_DOSHuge       0x06 // 16 bit FAT
    | >= 32mb
15457| #define PARTITION_NTFS         0x07
15458|
15459| #define PARTITION_DRDOS Huge     0xC4
15460| #define PARTITION_DRDOSExtended 0xC5
15461| #define PARTITION_DRDOSCompress 0xC6
15462|
15463| #define PARTITION_ConCurDOS    0xDB

```

```

15464|
15465| #define PARTITION_Netware286      0x64
15466| #define PARTITION_Netware386      0x65
15467|
15468| #define GetBeginSPT(Part) ((Part)->BeginSector & 0x3f)
15469| #define GetEndSPT(Part) ((Part)->EndSector & 0x3f)
15470| #define GetBeginCyl(Part) (((Part)->BeginSector &
    | 0xc0) << 2) + (Part)->BeginCyl)
15471| #define GetEndCyl(Part) (((Part)->EndSector & 0xc0) <<
    | 2) + (Part)->EndCyl)
15472|
15473|
15474| #pragma pack(1)
15475| typedef struct _PartInfo {
15476|     BYTE    Bootable;
15477|     BYTE    BeginHead;
15478|     BYTE    BeginSector;
15479|     BYTE    BeginCyl;
15480|     BYTE    SystemType;
15481|     BYTE    EndHead;
15482|     BYTE    EndSector;
15483|     BYTE    EndCyl;
15484|     ULONG   StartSector;
15485|     ULONG   NumberOfSectors;
15486| } tPartInfo;
15487|
15488| #pragma pack(1)
15489| // physical sector 0
15490| typedef struct _MBR {
15491|     BYTE    Code[0x1b8];
15492|     ULONG   SerialNumber;
15493|     BYTE    Dirty;
15494|     BYTE    Unknown;
15495|     tPartInfo  Part[4];
15496|     USHORT   Signature;
15497| } tMBR, *pMBR;
15498|
15499|
15500| #pragma pack()
15501|
15502|
15503|
15504| File Listing: mytypes.h
15505|
15506| typedef unsigned __int64 ULONGLONG;
15507|
15508| typedef struct _ULI {
15509|     union {
15510|         struct {
15511|             DWORD LowPart;

```

```

15512|         DWORD HighPart;
15513|     };
15514|     ULONGLONG QuadPart;
15515| };
15516| } ULI, *pULI;
15517|
15518| #define GetByte(Buffer,Offset)
15519| | ((BYTE*)(Buffer))[Offset]
15520| #define PutByte(Buffer,Offset,Byte) { \
15521| | ((BYTE*)(Buffer))[Offset] = ((Byte) & 0xff);\
15522| }
15523| #define GetChar(Buffer,Offset) ((signed
15524| | char*)(Buffer))[Offset]
15525| #define PutChar(Buffer,Offset,Char) { \
15526| | ((BYTE*)(Buffer))[Offset] = ((Char) & 0xff);\
15527| }
15528| #define GetWord(Buffer,Offset) *((unsigned
15529| | short*)&((BYTE*)(Buffer))[Offset])
15530| #define PutWord(Buffer,Offset,Word) { \
15531| | ((BYTE*)(Buffer))[Offset] = ((Word) &
15532| | 0xff);\
15533| | ((BYTE*)(Buffer))[Offset + 1] = (((Word) >> 8) &
15534| | 0xff);\
15535| }
15536| #define GetShort(Buffer,Offset) *(( signed
15537| | short*)&((BYTE*)(Buffer))[Offset])
15538| #define PutShort(Buffer,Offset,Short) { \
15539| | ((BYTE*)(Buffer))[Offset] = ((Short) &
15540| | 0xff);\
15541| | ((BYTE*)(Buffer))[Offset + 1] = (((Short) >> 8) &
15542| | 0xff);\
15543| }
15544| #define GetUTriByte(Buffer,Offset) ((*((unsigned
15545| | long*)&((BYTE*)(Buffer))[Offset])) & 0x00ffffff)
15546| // signed TriByte needs to be handled differently
15547| // this doesnt work...
15548| // #define GetTriByte(Buffer,Offset)
15549| | (GetUTriByte(Buffer,Offset) < 0x800000 ?
15550| | GetUTriByte(Buffer,Offset) : (signed
15551| | long)(GetUTriByte(Buffer,Offset) | 0xFF000000))

```

```

15544|
15545| // needs to be a function because we have to sign
    | extend the long
15546| signed long GetTriByte(PVOID Buffer,unsigned long
    | Offset);
15547|
15548|
15549| #define PutTriByte(Buffer,Offset,TriByte) { \
15550|     | ((BYTE*)(Buffer))[Offset]  = ((TriByte)    &
    | 0xff);\
15551|     | ((BYTE*)(Buffer))[Offset + 1] = (((TriByte) >> 8) &
    | 0xff);\
15552|     | ((BYTE*)(Buffer))[Offset + 2] = (((TriByte) >> 16) &
    | 0xff);\
15553|     }
15554| #define GetLong(Buffer,Offset)    *(( signed
    | long*)&((BYTE*)(Buffer))[Offset])
15555| #define PutLong(Buffer,Offset,Long) { \
15556|     | ((BYTE*)(Buffer))[Offset]  = ((Long)    &
    | 0xff);\
15557|     | ((BYTE*)(Buffer))[Offset + 1] = (((Long) >> 8) &
    | 0xff);\
15558|     | ((BYTE*)(Buffer))[Offset + 2] = (((Long) >> 16) &
    | 0xff);\
15559|     | ((BYTE*)(Buffer))[Offset + 3] = (((Long) >> 24) &
    | 0xff);\
15560|     }
15561| #define GetULong(Buffer,Offset)    *((unsigned
    | long*)&((BYTE*)(Buffer))[Offset])
15562|
15563|
15564| typedef char string8;
15565| #define LONGTOBYTE(x) ((BYTE)((x) & 0xff))
15566|
15567| typedef struct _BYTE_ARRAY {
15568|     CHAR Data[1024];
15569| } BYTE_ARRAY, *PBYTE_ARRAY;
15570|
15571| typedef struct _WORD_ARRAY {
15572|     WORD Data[1024 / 2];
15573| } WORD_ARRAY, *PWORD_ARRAY;
15574|
15575| typedef struct _ULONG_ARRAY {

```

```

15576|    ULONG Data[1024 / 4];
15577| } ULONG_ARRAY, *PULONG_ARRAY;
15578|
15579| typedef struct _ULARGE_ARRAY {
15580|     ULARGE_INTEGER Data[1024 / 8];
15581| } ULARGE_ARRAY, *PULARGE_ARRAY;
15582|
15583|
15584|
15585| File Listing: ntfs.c
15586|
15587| #define UNICODE
15588| #define _UNICODE
15589| #include <stdio.h>
15590| #include <stdlib.h>
15591| #include <string.h>
15592| #include <windows.h>
15593| #include <windowsx.h>
15594| #include <winioctl.h>
15595| // #include <crtdbg.h>           // _ASSERTE
15596| #include <assert.h>
15597| #define _ASSERTE assert
15598| #include <tchar.h>
15599| #include <conio.h>
15600| #include <io.h>
15601|
15602| #include "mytypes.h"
15603| #include "dasd.h"
15604| #include "fs.h"
15605| #include "ntfs.h"
15606| #include "dump.h"
15607| #include "safemem.h"
15608| #include "psm.h"
15609| #include "volume.h"
15610| #include "dlog.h"
15611|
15612| #include <undoc.h>
15613|
15614| // void PSM_LogDebugInfoW( const WCHAR *fmt,...);
15615| DWORD ExceptionFilter( EXCEPTION_POINTERS *ep );
15616|
15617| // todo containing record??
15618| //     generic link point
15619| //     understand the statement ; starting macros. If
15620| //         | to protect assignment they fail if part of further
15621| //         | expression.
15622| //     write this better
15623| //     make todo part of warning!!

```



```

15624|
15625|
15626| /*
15627|  TODO:
15628|  Reading the $Attrdef file for attribute IDs
15629|  Log file structures
15630|  Add/get b+ tree routines for directory parsing
15631|  POSIX parsing
15632|
15633| Side note: It seems that the files below are protected
      | with a security descriptor and that
15634| is why the files do not show up in the directory list.
      | When you try to access the file, you
15635| get "Access denied" I created a file called $Test with
      | the same attributes as $mft, and it
15636| showed up in the dir list, the mft of the 2 files were
      | pretty much the same. If you do a
15637| 'dir $mft /ah' the file does show up.. maybe the NTFS
      | file system justs parses them out when
15638| making the directory list?
15639|
15640| When doing an index, we will need the following files
      | to satisfy a mount
15641|  $Mft          At least first 16 entries (32k-64k)
      | Mine is 145MB
15642|  $MftMirr      4k
15643|  $LogFile      ~3 MB is the default, but can be
      | bigger
15644|  $Volume       0k
15645|  $AttrDef      36000 bytes
15646|  $Root         Need INDEX_ROOT_ATTR,
      | INDEX_ALLOCATION_ATTR, and BITMAP_ATTR
15647|  $Bitmap       1 bit per cluster on the volume
15648|  $Boot         8k
15649|  $BadClus      0k,
15650|              The VCN To LCN mappings correspond
      | to the bad sectors (thus the LCN of
15651|              VCN X is a bad sector) so do not
      | access this file
15652|  $Quota        0k
15653|  $Upcase       128k
15654|
15655|
15656|
15657| */
15658|
15659| // needs to be a function because we have to sign
      | extend the long
15660|
15661| signed long GetTriByte(PVOID Buffer,unsigned long

```

```

    | Offset)
15662| {
15663|     unsigned long SL = *(( signed
    | long*)&((BYTE*)(Buffer))[(Offset)]) & 0x00ffffff;
15664|     if(SL>0x7ffff) SL-=0x1000000;
15665|     return SL;
15666| }
15667|
15668|
15669| /*
15670|     This routine came from the linux driver and has
    | been modified slightly
15671|     TODO: rewrite this in a readable and understandable
    | form. See pages 61-67 of
15672|     Helen's books on a description of the algorithm.
15673|
15674|     Given a compressed buffer of src, decompresses it
    | into dest coping a max of Size
15675| */
15676| void NTFS-DecompressBuffer(unsigned char *dest,
    | unsigned char *src, ULONGLONG Size)
15677| {
15678|     int head,comp;
15679|     int copied=0;
15680|     unsigned char *stop;
15681|     int bits;
15682|     int tag=0;
15683|     int clear_pos;
15684|     while(1)
15685|     {
15686|         head = GetWord(src,0) & 0xFFF;
15687|         /* high bit indicates that compression was
    | performed */
15688|         comp = GetByte(src,1) & 0x80;
15689|         //comp = (head == 0xFFF);
15690|         src += 2;
15691|         stop = src+head;
15692|         bits = 0;
15693|         clear_pos=0;
15694|         if(head==0)
15695|             /* block is not used */
15696|             return; /* FIXME: copied */
15697|         if(!comp) /* uncompressible */
15698|         {
15699|             memcpy(dest,src,0x1000);
15700|             dest+=0x1000;
15701|             copied+=0x1000;
15702|             src+=0x1000;
15703|             if(Size==copied)
15704|                 return;

```

```

15705|         continue;
15706|     }
15707|     while(src<=stop)
15708|     {
15709|         if(clear_pos>4096)
15710|         {
15711|             DLOG((TEXT("Error 1 in
| decompress\n"))));
15712|             return;
15713|         }
15714|         if(!bits){
15715|             tag=GetByte(src,0);
15716|             bits=8;
15717|             src++;
15718|             if(src>stop)
15719|                 break;
15720|         }
15721|         if(tag & 1){
15722|             int i,len,delta,code,lmask,dshift;
15723|             code = GetWord(src,0);
15724|             src+=2;
15725|             if(!clear_pos)
15726|             {
15727|                 DLOG((TEXT("Error 2 in
| decompress\n"))));
15728|                 return;
15729|             }
15730|             | for(i=clear_pos-1,lmask=0xFFF,dshift=12;i>=0x10;i>=1)
15731|             {
15732|                 lmask >>= 1;
15733|                 dshift--;
15734|             }
15735|             delta = code >> dshift;
15736|             len = (code & lmask) + 3;
15737|             for(i=0; i<len; i++)
15738|             {
15739|                 | dest[clear_pos]=dest[clear_pos-delta-1];
15740|                 clear_pos++;
15741|                 copied++;
15742|                 if(copied==Size)
15743|                     return;
15744|             }
15745|         }else{
15746|             dest[clear_pos++]=GetByte(src,0);
15747|             src++;
15748|             copied++;
15749|             if(copied==Size)
15750|                 return;

```

```

15751|     }
15752|     tag>=>1;
15753|     bits--;
15754|     }
15755|     dest+=clear_pos;
15756| }
15757| }
15758|
15759|
15760| /*
15761|    Given a VCN for File, will return its LCN. The
        | file must be non resident
15762|    returns -1 if an error occurs, Use GetLastError()
        | to get the error
15763| */
15764| ULONGLONG NTFS_VCNToLCN(PNTFS_File File, ULONGLONG VCN
        | )
15765| {
15766|     ULONG Run=0;
15767|     PATTRIBUTE
        | Attribute=NTFS_GetAttributeByName(File->VolumeInfo,File-
        | >Mft,File->Type,File->AttrName);
15768|
15769|     if(Attribute) {
15770|         if(Attribute->NotResident) {
15771|             while(
                | (Run<File->VCNToLCNMappingTableSize) &&
15772|             | (VCN>=File->VCNToLCNMappingTable[Run].Length)) {
15773|                 VCN -=
                | File->VCNToLCNMappingTable[Run++].Length;
15774|             }
15775|             if(Run==File->VCNToLCNMappingTableSize) {
15776|                 SetLastError(ERROR_SECTOR_NOT_FOUND);
15777|                 return (ULONGLONG)-1;
15778|             } else
15779|             | if(File->VCNToLCNMappingTable[Run].Cluster==(ULONGLONG)-
                | 1)
15780|                 return (ULONGLONG)-1;
15781|             else
15782|                 return
                | File->VCNToLCNMappingTable[Run].Cluster+VCN;
15783|         } else {
15784|             // resident
15785|             SetLastError(ERROR_INVALID_PARAMETER);
15786|             return (ULONGLONG)-1;
15787|         }
15788|     } else {
15789|         // no attribute found

```

```

15790|     SetLastError(ERROR_NO_ATTRIBUTE_FOUND);
15791|     return (ULONGLONG)-1;
15792| }
15793| }
15794|
15795| /*
15796|  a fixup technique : the last word of each sector
    | (called a fixup) of a
15797|  structure's record should end with the word at
    | offset <n> of the first
15798|  sector, and if it is the case, must be replaced
    | with the words following
15799|  <n>. The value of <n> and the number of fixups is
    | taken from the fields
15800|  at the offsets 4 and 6.
15801| */
15802| int NTFS_FixupRecord ( pVolumeInfo VolumeInfo, PVOID
    | Record )
15803| {
15804|     WORD Count,Offset,Fixup,Start;
15805|
15806|     Start = GetWord(Record, 4);
15807|     Count = GetWord(Record, 6);
15808|     Fixup = GetWord(Record, Start);
15809|
15810|     // the first fixup is the check key, so skip it.
15811|     Start += sizeof(WORD);
15812|     Offset =
        | VolumeInfo->NtfsBootSector->BytesPerSector-sizeof(WORD);
15813|     Count--;
15814|
15815|     while(Count--) {
15816|         // the field doesnt contain the key, must be
        | bad...
15817|         if (GetWord(Record,Offset)!=Fixup)
15818|             return 0;
15819|         PutWord(Record,Offset,GetWord(Record,Start));
15820|         Offset +=
            | VolumeInfo->NtfsBootSector->BytesPerSector;
15821|         Start += sizeof(WORD);
15822|     }
15823|     return 1;
15824| }
15825|
15826| int NTFS_UnfixupRecord ( pVolumeInfo VolumeInfo, PVOID
    | Record )
15827| {
15828|     WORD Count, Offset, Fixup, Start;
15829|
15830|     Start = GetWord(Record, 4);

```

```

15831|   Count = 1 +
      | (USHORT)VolumelInfo->NtfsBootSector->MFTRecordSize;
15832|   Fixup = 1 + GetWord(Record, Start);
15833|   PutWord(Record,Start,Fixup);
15834|
15835|   Start += sizeof(WORD);
15836|   Offset = VolumelInfo->NtfsBootSector->BytesPerSector
      | - sizeof(WORD);
15837|
15838|   while(--Count) {
15839|       PutWord (Record,Offset,Fixup);
15840|       Offset +=
      | VolumelInfo->NtfsBootSector->BytesPerSector;
15841|       Start += sizeof(WORD);
15842|   }
15843|
15844|   return 1;
15845| }
15846|
15847| /*
15848|   This decompresses a run, and returns it.
15849|   Runs are stored as: 0xCL where L is the size of the
      | length and C is the size of the
15850|   cluster. So if we have a byte stream of 22 12 34
      | 56 78, the length is a word (1234) and
15851|   the cluster is a word (5678). if a stream of 21 01
      | 12 34, the length is a byte (01) and
15852|   the cluster is a word (1234). A Length of 0 ends
      | the run. A Cluster of 0 indicates a
15853|   sparse run. The Cluster is signed and added to the
      | previous cluster number. negative
15854|   clusters go back from the current position. The
      | first run starts at cluster 0.
15855|   Length is unsigned.
15856|
15857|   TODO: added Types 5-8
15858| */
15859|
15860| #define GetLengthLength(Type) ((Type) & 0x0f)
15861| #define GetClusterLength(Type) ((Type) >> 4)
15862| int GetRun ( PVOID DataRun, ULONG *Offset,
      | ULARGE_INTEGER *Cluster, ULARGE_INTEGER *Length, ULONG
      | *Sparse )
15863| {
15864|   CHAR Type = GetByte(DataRun,(*Offset));
15865|   (*Sparse) = 0;
15866|
15867|   if(!Type)
15868|       return 0;
15869|

```

```

15870|    (*Offset)++;
15871|    switch(GetLengthLength(Type)) {
15872|        case 1: Length->QuadPart =
15873|            | GetByte(DataRun,(*Offset));
15874|            break;
15875|        case 2: Length->QuadPart =
15876|            | GetWord(DataRun,(*Offset));
15877|            break;
15878|        case 3: Length->QuadPart =
15879|            | GetUTriByte(DataRun,(*Offset));
15880|            break;
15881|        case 4: Length->QuadPart =
15882|            | GetULong(DataRun,(*Offset));
15883|            break;
15884|        default:
15885|            DLOG((TEXT("Unable to decompress run length
15886| of %d\n"),GetLengthLength(Type)));
15887|            break;
15888|    }
15889|    (*Offset) += GetLengthLength(Type);
15890|
15891|    switch(GetClusterLength(Type)) {
15892|        case 0: (*Sparse) = 1;
15893|            break;
15894|        case 1: Cluster->QuadPart +=
15895|            | GetChar(DataRun,(*Offset));
15896|            break;
15897|        case 2: Cluster->QuadPart +=
15898|            | GetShort(DataRun,(*Offset));
15899|            break;
15900|        case 3: Cluster->QuadPart +=
15901|            | GetTriByte(DataRun,(*Offset));
15902|            break;
15903|        case 4: Cluster->QuadPart +=
15904|            | GetLong(DataRun,(*Offset));
15905|            break;
15906|        default:
15907|            DLOG((TEXT("Unable to decompress run length
15908| of %d\n"),GetClusterLength(Type)));
15909|            break;
15910|    }
15911|    (*Offset) += GetClusterLength(Type);
15912|    return 1;
15913| }
15914|
15915|
15916| /*
15917| Opens file with its Mft Entry number. Usually only
15918| | used for directories and well known
15919| | Ids (FILE_MFT, FILE_BITMAP, FILE_ROOT, etc..) Type

```

```

    | is the attribute(stream) to open.
15909| */
15910| PNTFS_File NTFS_OpenFileByNumber ( pVolumeInfo
    | VolumeInfo, ULONGLONG MftEntryNum, ULONG Type, WCHAR
    | *AttrName )
15911| {
15912|     ULONG Err=0;
15913|     PNTFS_File NtfsFile=SAFE_Alloc(sizeof(NTFS_File));
15914|
15915|     if(NtfsFile) {
15916|
        | NtfsFile->Mft=SAFE_Alloc(NTFS_MftByteSize(VolumeInfo));
15917|
15918|         if(NtfsFile->Mft) {
15919|             // buffer for file io
15920|             NtfsFile->Buffer =
        | SAFE_Alloc(NTFS_ClusterSizeInBytes(VolumeInfo));
15921|             if(NtfsFile->Buffer) {
15922|                 // read in the file Mft
15923|                 Err = NTFS_ReadMftEntryNumber(
        | VolumeInfo, MftEntryNum, NtfsFile->Mft );
15924|                 if(!Err) {
15925|                     // get the clusters for the file
15926|                     PATTRIBUTE Attribute =
        | NTFS_GetAttributeByName( VolumeInfo, NtfsFile->Mft,
        | Type, AttrName );
15927|                     if(Attribute) {
15928|                         if(Attribute->NotResident) {
15929| //old line replaced by sub function to allow extension
        | data runs   PVOID DataRun = NTFS_GetDataPointer(
        | Attribute );
15930|                         PVOID DataRun =
        | NTFS_GetDataRun(Attribute, NtfsFile, VolumeInfo,
        | MftEntryNum, Type, AttrName ) ;
15931|
15932|                         NtfsFile->CompressedBuffer
        | = NULL;
15933|
        | NtfsFile->UnCompressedBuffer = NULL;
15934|                         if(Attribute->Compressed) {
15935|
        | NtfsFile->CompressedBuffer =
        | SAFE_Alloc(NTFS_ClusterSizeInBytes(VolumeInfo)*16);
15936|
        | NtfsFile->UnCompressedBuffer =
        | SAFE_Alloc(NTFS_ClusterSizeInBytes(VolumeInfo)*16);
15937|
        | if((!NtfsFile->CompressedBuffer) ||
        | (!NtfsFile->UnCompressedBuffer)) {
15938|

```



```

    | if(NtfsFile->CompressedBuffer)
15939|
    | SAFE_Free(NtfsFile->CompressedBuffer);
15940|
    | if(NtfsFile->UnCompressedBuffer)
15941|
    | SAFE_Free(NtfsFile->UnCompressedBuffer);
15942|           DataRun=NULL;
15943|           }
15944|       }
15945|
15946|           if(DataRun) {
15947|
    | NtfsFile->VCNTToLCNMappingTableSize = GetDataRunLength(
    | DataRun,Attribute->NonResidentData.LastVCN.QuadPart-Attr
    | ivate->NonResidentData.StartingVCN.QuadPart+1 );
15948|           //DLOG((TEXT("VCNTToLCN
    | mapping table
    | size=%d\n"),NtfsFile->VCNTToLCNMappingTableSize*sizeof(DA
    | TA_RUN)));
15949|           // TODO: malloc needs a
    | unsigned int, im passing in an unsigned __int64
15950|
    | NtfsFile->VCNTToLCNMappingTable =
    | SAFE_Alloc(NtfsFile->VCNTToLCNMappingTableSize*sizeof(DAT
    | A_RUN));
15951|
15952|
    | if(NtfsFile->VCNTToLCNMappingTable) {
15953|           if((Err =
    | MakeVCNTToLCNMapping( DataRun,
    | NtfsFile->VCNTToLCNMappingTable,
    | NtfsFile->VCNTToLCNMappingTableSize))==0) {
15954|           NtfsFile->Type
    | = Type;
15955|
    | NtfsFile->MftEntryNum = MftEntryNum;
15956|
    | NtfsFile->VolumeInfo = VolumeInfo;
15957|           // protect
    | resources.
15958|           SAFE_ReadOnly(
    | NtfsFile );
15959|           SAFE_ReadOnly(
    | NtfsFile->Mft );
15960|           SAFE_ReadOnly(
    | NtfsFile->VCNTToLCNMappingTable );
15961|           SAFE_ReadOnly(
    | NtfsFile->Buffer );
15962|           //clumsy way

```

[illegible]

```

    | found in file!\n"),Type));
15994|
    | SetLastError(ERROR_NO_ATTRIBUTE_FOUND);
15995|         }
15996|     } else {
15997|         DLOG((TEXT("Error %08x reading
    | Mft\n"),Err));
15998|         SetLastError(Err);
15999|     }
16000|     SAFE_Free(NtfsFile->Buffer);
16001| } else {
16002|     DLOG((TEXT("Out of memory\n")));
16003|     SetLastError(ERROR_OUTOFMEMORY);
16004| }
16005|
16006|     SAFE_Free(NtfsFile->Mft);
16007| } else {
16008|     DLOG((TEXT("Error out of memory\n")));
16009|     SetLastError(ERROR_OUTOFMEMORY);
16010| }
16011|     SAFE_Free(NtfsFile);
16012| } else {
16013|     DLOG((TEXT("Error out of memory\n")));
16014|     SetLastError(ERROR_OUTOFMEMORY);
16015| }
16016| return NULL;
16017| }
16018|
16019| /*
16020|  makes a string all uppercase based on the unicode
    | translation table.
16021|
16022|  Given unicode char Lower, this provides the
    | uppercase translation table so
16023|  Upper = UpcaseTable[Lower];
16024| */
16025| WCHAR *NTFS_MakeUpperCase( pVolumeInfo VolumeInfo,
    | WCHAR *Str, ULONG Length )
16026| {
16027|     WCHAR *p=Str;
16028|     ULONG C=0;
16029|     while(C++<Length) {
16030|         *p++ = VolumeInfo->UpcaseTable[*p];
16031|     }
16032|     return Str;
16033| }
16034|
16035| /*
16036|  Meat and potatoes of ntfs directory parsing
    | routine.

```

```

16037|
16038| Returns the MftEntry of the file or -1, use
    | GetLastError() to get error code
16039| */
16040| ULONGLONG NTFS_RecurseDir( pVolumeInfo VolumeInfo,
    | ULONGLONG MftEntry, WCHAR *Name )
16041| {
16042|     PNTFS_File IRFile=NTFS_OpenFileByNumber(
    | VolumeInfo, MftEntry, INDEX_ROOT_ATTR, NULL );
16043|     ULONG Err=0;
16044|     if(IRFile) {
16045|         PNTFS_File IFile = NTFS_OpenFileByNumber(
    | VolumeInfo, MftEntry, INDEX_ALLOCATION_ATTR, NULL );
16046|         ULONGLONG IRSize = NTFS_GetFileSize(IRFile);
16047|         PINDEX_ROOT IR =
    | SAFE_Alloc((ULONG)IRSize);
16048|
16049|         //DumpMft(VolumeInfo,IFile->Mft);
16050|         if(IR) {
16051|             PINDEX_ENTRY IE =
    | (PINDEX_ENTRY)(IR+1);
16052|             Err = NTFS_ReadFile( IRFile, 0, IRSize, IR
    | );
16053|             SAFE_ReadOnly(IR);
16054|
16055|             if(!Err) {
16056|                 ULONGLONG IASize=0;
16057|                 PINDEX_ALLOCATION IA=NULL;
16058|                 ULONG NameLength;
16059|                 WCHAR *p=Name;
16060|                 BOOLEAN DoExit=FALSE;
16061|                 ULONGLONG DirMftEntry;
16062|                 PFILENAME FN =
    | NTFS_GetDataPointer((PATRIBUTE)NTFS_GetAttributeByName(
    | IRFile->VolumeInfo, IRFile->Mft, FILENAME_ATTR, NULL
    | ));
16063|
16064|                 NameLength = FN->FileNameLength;
16065|                 if(IFile) {
16066|                     IASize = IR->SizeOfIndex;
16067|                     IA =
    | SAFE_Alloc((ULONG)IASize);
16068|                     SAFE_ReadOnly(IA);
16069|                 }
16070|
16071| #if 0
16072|                 DLOG((TEXT("Mft Entry=%I64x,
    | FileName='%s'\n"),MftEntry,NameLength,NameLength,FN-
    | >FileName));
16073|                 DLOG((TEXT("SizeOfIndex=%d,

```

```

    | NumberOfClustersPerIndex=%d\n"),IR->SizeOfIndex,
    | IR->NumberOfClustersPerIndex));
16074|         DLOG((TEXT("Unknown 1=%08x, 2=%082,
    | 3=%08x, 4=%08x,
    | 5=%08x\n"),IR->Unknown1,IR->Unknown2,IR->Unknown3,IR->Un
    | known4,IR->Unknown5));
16075| #endif
16076|
16077|         // get the length of the path we are
    | interested in
16078|         // ie if passed
    | WINNT\SYSTEM32\CONFIG\software.log we only
16079|         // want <WINNT> as we will
    | recurse into the others
16080|         // assuming the directories exist
16081|         while((*p!=L'\0') && (*p!=L'\')) {
16082|             p++;
16083|         }
16084|         NameLength=p-Name;
16085|
    | NTFS_MakeUpperCase(VolumeInfo,Name,NameLength);
16086| #if 0
16087|         DLOG((TEXT("Searching for
    | '%-*.s\n"),NameLength,NameLength,Name));
16088| #endif
16089|
16090|         while(!DoExit) {
16091| #if 0
16092|             if(IE->DataSize) {
16093|                 DLOG((
16094|                     TEXT("'%-*.s\n")
16095|                     TEXT("
    | DirectoryMft=%012l64x, SequenceNumber=%04l64x\n")
16096|                     TEXT(" EntrySize=%04x,
    | DataSize=%04x (%04x), Flags=%08x\n")
16097|                     TEXT("
    | MyDirectoryEntry=%012l64x, MySequenceNumber=%04l64x\n")
16098|                     TEXT("
    | AllocatedAttributeSize=%016l64x,
    | AttributeSize=%016l64x\n")
16099|                     TEXT(" Attributes=%016l64x
    | (%s)\n"),
16100|                     IE->FileNameLength,
16101|                     IE->FileNameLength,
16102|                     IE->FileName,
16103|                     IE->DirectoryMft,
    | IE->SequenceNumber, IE->EntrySize,
16104|
    | IE->DataSize,IE->EntrySize-IE->DataSize, IE->Flags,
16105|                     IE->MyDirectoryEntry,

```

```

    | IE->MySequenceNumber,
16106|
    | IE->AllocatedAttributeSize, IE->AttributeSize,
    | IE->Attributes,
    | GetDosAttributes(IE->Attributes.LowPart));
16107|         } else {
16108|             DLOG((
16109|                 TEXT("<Last Entry>\n")
16110|                 TEXT("
    | DirectoryMft=%012l64x, SequenceNumber=%04l64x\n")
16111|                 TEXT(" EntrySize=%04x,
    | DataSize=%04x, Flags=%08x\n")
16112|                 TEXT("
    | MyDirectoryEntry=%012l64x,
    | MySequenceNumber=%04l64x\n"),
16113|                 IE->DirectoryMft,
    | IE->SequenceNumber, IE->EntrySize,
16114|                 IE->DataSize,
    | IE->Flags,
16115|                 IE->MyDirectoryEntry,
    | IE->MySequenceNumber ));
16116|         }
16117| #endif
16118|         if(IE->Flags & FLAGS_INDEX_LAST) {
16119|             // handle as special case, so
    | no file name may be here.
16120|             // this is usually a dummy
    | entry
16121|             if(IE->DataSize != 0) {
16122|                 DoExit = TRUE;
16123|                 goto DoFileName;
16124|             }
16125|             goto DoSubNodes;
16126|         } else {
16127| DoFileName:
16128|             // keys are based on filenames
16129|             SAFE_ReadWrite(IE);
16130|
    | NTFS_MakeUpperCase(VolumeInfo,IE->FileName,IE->FileNameL
    | ength);
16131|             SAFE_ReadOnly(IE);
16132|
16133|             // for faster access, only
    | compare if the lengths are the same
16134|             if(IE->FileNameLength ==
    | NameLength) {
16135|
    | if(wcsncmp(Name,IE->FileName,NameLength)==0) {
16136|                 // Yeah! The file has
    | been found!

```

```

16137|                // now see if we need
    | to recurse into ourselves again.
16138|                if(*p!=L'\0') {
16139|                    if((*p=='\\') &&
    | (*p+1)==L'\0')) goto
16140|                        DoAsFile;
16141|
    | if(IE->Attributes.LowPart & FILENAME_DIRECTORY) {
16142|                // user is
    | treating this as a directory
16143|                DirMftEntry =
    | IE->DirectoryMft;
16144|                // we no longer
    | need this data so free them
16145|                if(IAFile) {
16146|                    | SAFE_Free(IA);
16147|                    | NTFS_CloseFile(IAFile);
16148|                }
16149|                SAFE_Free(IR);
16150|
    | NTFS_CloseFile(IRFile);
16151|                return
    | NTFS_RecurseDir( VolumeInfo, DirMftEntry, p+1 );
16152|                } else {
16153|                // user is
    | treating this as a directory, but
16154|                // it isnt
16155|                Err =
    | ERROR_PATH_NOT_FOUND;
16156|                DoExit = TRUE;
16157|                }
16158|                } else {
16159| DoAsFile:
16160|                // User is treating
    | this as a file, If a directory,
16161|                // user could be
    | opening it for direct access. This
16162|                // method is used
    | for directory walking
16163|                DirMftEntry =
    | IE->DirectoryMft;
16164|                // free the data
    | and return
16165|                if(IAFile) {
16166|                    SAFE_Free(IA);
16167|                    | NTFS_CloseFile(IAFile);
16168|                }

```

```

16169|             SAFE_Free(IR);
16170|
16171|             return DirMftEntry;
16172|         }
16173|         // should never get
16174|         | here, but if we do
16175|         DoExit = TRUE;
16176|     } else
16177|         goto DoCompare;
16178|     } else {
16179| DoCompare:
16180|
16181|         if(wcsncmp(Name,IE->FileName,min(IE->FileNameLength,Name
16182|         | Length))<0) {
16183| LessThan:
16184| DoSubNodes:
16185|         // lets take the low
16186|         | road
16187|         if(IE->Flags &
16188|         | FLAGS_INDEX_SUBNODES) {
16189|         // impossible to
16190|         | have subnodes and no allocation index
16191|         if(IAFile) {
16192|         ULONGLONG VCN =
16193|         | NTFS_GetSubNodesVCN(IE);
16194|
16195|         | SAFE_ReadWrite(IA);
16196|         Err =
16197|         | NTFS_ReadFile( IAFile,
16198|         | VCN*NTFS_ClusterSizeInBytes(VolumeInfo), IASize, IA );
16199|
16200|         | //_ASSERT(IA->AllocatedLength<IASize);
16201|
16202|         if((Err==0) &&
16203|         | (IA->Signature=='XDNI') && (NTFS_FixupRecord(
16204|         | VolumeInfo, IA ))) {
16205|
16206|         | SAFE_ReadOnly(IA);
16207|
16208|         IE =
16209|         | (PINDEX_ENTRY)&((BYTE*)IA)[0x18+IA->HeaderSize];
16210|
16211|         #if 0
16212|
16213|         | DLOG((TEXT("Reading VCN %I64d\n"),VCN));
16214|
16215|         | DLOG((TEXT("Unknown = %016I64x, VCNOfBuffer=%I64d,
16216|         | HeaderSize=%04x\n"),
16217|         | IA->Unknown,
16218|

```



```

    | IA->VCNOOfBuffer,
16199|
    | IA->HeaderSize));
16200|
    | DLOG((TEXT("Unknown2=%04x, Length=%08x,
    | AllocatedLength=%08x, Unknown3=%08x\n"),
16201|
    | IA->Unknown2,
16202|
    | IA->Length,
16203|
    | IA->AllocatedLength,
16204|
    | IA->Unknown3,
16205|
    | IA->Fixup));
16206| #endif
16207|                 } else {
16208|                 DoExit =
    | TRUE;
16209|                 Err =
    | ERROR_DISK_CORRUPT;
16210|                 }
16211|                 } else {
16212|                 DoExit = TRUE;
16213|                 Err =
    | ERROR_DISK_CORRUPT;
16214|                 }
16215|                 } else {
16216|                 // no files found
16217|                 Err =
    | ERROR_FILE_NOT_FOUND;
16218|                 DoExit = TRUE;
16219|                 }
16220|                 } else
16221|
    | if(wcsncmp(Name,IE->FileName,min(IE->FileNameLength,Name
    | Length))>0) {
16222| GreaterThan:
16223|                 // lets take the high
    | road.
16224|
    | ((BYTE*)IE)+=NTFS_IndexEntrySize(IE);
16225|                 } else {
16226|                 // at this point, we
    | have the following condition
16227|                 // 1.
    | NameLength!=FN->FileNameLength
16228|                 // 2. First X chars of
    | Y match Z

```

```

16229|                // example:
16230|                //   FN->FileName =
16231|                | 'SYS'   Length=3
16232|                //   Name      =
16233|                | 'SYSTEM' Length=6
16234|                | if(NameLength>IE->FileNameLength)
16235|                |     goto GreaterThan;
16236|                |     else
16237|                |     goto LessThan;
16238|                |     }
16239|                |     }
16240|                | } // not last node
16241|                | } // while(!DoExit)
16242|                | if(IA)
16243|                |     SAFE_Free(IA);
16244|                | } else {
16245|                |     DLOG((TEXT("Error %08x reading from
16246|                | Index root\n"),Err));
16247|                |     }
16248|                |     SAFE_Free(IR);
16249|                | } else {
16250|                |     DLOG((TEXT("Error, out of memory\n")));
16251|                |     Err = ERROR_OUTOFMEMORY;
16252|                |     }
16253|                |     if(IAFile)
16254|                |     NTFS_CloseFile(IAFile);
16255|                |     NTFS_CloseFile(IRFile);
16256|                | } else {
16257|                |     // user passed in a filename as a path, example
16258|                |     | : \winnt\system32\calc.exe\file
16259|                |     DLOG((TEXT("Not a directory\n")));
16260|                |     Err = ERROR_PATH_NOT_FOUND;
16261|                |     }
16262|                |     SetLastError(Err);
16263|                |     return (ULONGLONG)-1;
16264|                | }
16265|                | }
16266|                | ULONGLONG DirWalkDir( tVolumeInfo *VolumeInfo,
16267|                |     | PNTFS_File IAFile, ULONGLONG IASize, PINDEX_ALLOCATION
16268|                |     | IA, PINDEX_ENTRY IE, int Level, ULONGLONG *Count )
16269|                | {
16270|                |     ULONG Err;
16271|                |
16272|                |     //DLOG((TEXT("%-*.*s
16273|                |     | VCN=%016I64x\n"),Level*3,Level*3,TEXT("
16274|                |     | "),MyVCN));
16275|                |     while(1) {
16276|                |         if(IE->Flags & FLAGS_INDEX_SUBNODES) {
16277|                |             if(IAFile) {

```

```

16270|         ULONGLONG MyVCN =
| IA->VCNOfBuffer.QuadPart;
16271|         ULONGLONG VCN =
| NTFS_GetSubNodesVCN(IE);
16272|         SAFE_ReadWrite(IA);
16273|         Err = NTFS_ReadFile( IFile,
| VCN*NTFS_ClusterSizeInBytes(VolumeInfo), IASize, IA );
16274|         _ASSERTE(IA->AllocatedLength<IASize);
16275|
16276|         if((Err==0) && (IA->Signature=='XDNI')
| && (NTFS_FixupRecord( VolumeInfo, IA ))) {
16277|             PINDEX_ENTRY NewIE;
16278|             _ASSERTE(IA->VCNOfBuffer.QuadPart
| == VCN);
16279|             SAFE_ReadOnly(IA);
16280|             NewIE =
| (PINDEX_ENTRY)&((BYTE*)IA)[0x18+IA->HeaderSize];
16281|             DirWalkDir( VolumeInfo, IFile,
| IASize, IA, NewIE, Level+1,Count );
16282|             // if not the root
16283|             if(MyVCN!=(ULONGLONG)-1) {
16284|                 // read our data again
16285|                 SAFE_ReadWrite(IA);
16286|                 Err = NTFS_ReadFile( IFile,
| MyVCN*NTFS_ClusterSizeInBytes(VolumeInfo), IASize, IA
| );
16287|                 _ASSERTE(IA->AllocatedLength<IASize);
16288|                 // already passed test above so
| we dont need to check again...
16289|                 NTFS_FixupRecord( VolumeInfo,
| IA );
16290|                 SAFE_ReadOnly(IA);
16291|             }
16292|             } else {
16293|                 SetLastError(ERROR_DISK_CORRUPT);
16294|                 return (ULONGLONG)-1;
16295|             }
16296|             } else {
16297|                 SetLastError(ERROR_DISK_CORRUPT);
16298|                 return (ULONGLONG)-1;
16299|             }
16300|         }
16301|
16302|         if(!(IE->Flags & FLAGS_INDEX_LAST)) {
16303|             if(IE->FileNameType != FILENAME_DOS)
16304| #if 1
16305|                 DLOG((TEXT("%04I64x: %012I64x
| %-*.s\n"),
16306|                     (*Count),

```

```

16307|         IE->DirectoryMft,
16308|         IE->FileNameLength,
16309|         IE->FileNameLength,
16310|         IE->FileName
16311|     ));
16312|
16313| #else
16314|         DLOG((
16315|             TEXT("%-*. *s ")
16316|             TEXT("DirectoryMft=%012l64x,
| SequenceNumber=%04l64x, ")
16317|             TEXT("EntrySize=%04x, DataSize=%04x
| (%04x), Flags=%08x, ")
16318|             TEXT("MyDirectoryEntry=%012l64x,
| MySequenceNumber=%04l64x, ")
16319|             | TEXT("AllocatedAttributeSize=%016l64x,
| AttributeSize=%016l64x, ")
16320|             TEXT("Attributes=%016l64x (%s), ")
16321|             TEXT("%-*. *s\n"),
16322|             Level*3,Level*3,TEXT("
| "),
16323|             IE->DirectoryMft,
| IE->SequenceNumber, IE->EntrySize,
16324|             | IE->DataSize,IE->EntrySize-IE->DataSize, IE->Flags,
16325|             IE->MyDirectoryEntry,
| IE->MySequenceNumber,
16326|             IE->AllocatedAttributeSize,
| IE->AttributeSize, IE->Attributes,
| GetDosAttributes(IE->Attributes.LowPart),
16327|             IE->FileNameLength,
16328|             IE->FileNameLength,
16329|             IE->FileName
16330|         ));
16331| #endif
16332|         (*Count)++;
16333|
16334|     } else
16335|         break;
16336|
16337|     ((BYTE*)IE)+=NTFS_IndexEntrySize(IE);
16338| }
16339| SetLastError(0);
16340| return (ULONGLONG)-1;
16341| }
16342|
16343| ULONGLONG DisplayFilesInDir( tVolumeInfo *VolumeInfo,
| ULONGLONG MftEntry )
16344| {

```

```

16345|   PNTFS_File IRFile=NTFS_OpenFileByNumber(
      | VolumeInfo, MftEntry, INDEX_ROOT_ATTR, NULL );
16346|   ULONG Err=0;
16347|   if(IRFile) {
16348|       PNTFS_File IAFFile = NTFS_OpenFileByNumber(
      | VolumeInfo, MftEntry, INDEX_ALLOCATION_ATTR, NULL );
16349|       ULONGLONG ISize = NTFS_GetFileSize(IRFile);
16350|       PINDEX_ROOT IR   =
      | SAFE_Alloc((ULONG)ISize);
16351|
16352|       //DumpMft(VolumeInfo,IAFile->Mft);
16353|       if(IR) {
16354|           PINDEX_ENTRY   IE =
      | (PINDEX_ENTRY)(IR+1);
16355|           Err = NTFS_ReadFile( IRFile, 0, ISize, IR
      | );
16356|           SAFE_ReadOnly(IR);
16357|
16358|           if(!Err) {
16359|               ULONGLONG   IASize;
16360|               PINDEX_ALLOCATION IA=NULL;
16361|               ULONGLONG   Count=0;
16362|
16363|               if(IAFile) {
16364|                   IASize = IR->SizeOfIndex;
16365|                   IA     =
      | SAFE_Alloc((ULONG)IASize);
16366|                   if(IA) {
16367|                       IA->VCNOOfBuffer.QuadPart =
      | (ULONGLONG)-1;
16368|                       SAFE_ReadOnly(IA);
16369|                   } else {
16370|                       Err = ERROR_OUTOFMEMORY;
16371|                   }
16372|               }
16373|
16374|               if (((IAFile!=NULL) && (IA!=NULL)) ||
16375|                   ((IAFile==NULL))) {
16376|                   DirWalkDir(VolumeInfo, IAFFile,
      | IASize, IA, IE, 0,&Count );
16377|                   DLOG((TEXT("Total files =
      | %!64d\n"),Count));
16378|               }
16379|
16380|               if(IA)
16381|                   SAFE_Free(IA);
16382|           } else {
16383|               DLOG((TEXT("Error %08x reading from
      | Index root\n"),Err));
16384|           }

```

```

16385|     SAFE_Free(IR);
16386| } else {
16387|     DLOG((TEXT("Error, out of memory\n")));
16388|     Err = ERROR_OUTOFMEMORY;
16389| }
16390| if(IAFile)
16391|     NTFS_CloseFile(IAFile);
16392|     NTFS_CloseFile(IRFile);
16393| } else {
16394|     // user passed in a filename as a path, example
16395|     | : \winnt\system32\calc.exe\file
16396|     DLOG((TEXT("Not a directory\n")));
16397|     Err = ERROR_PATH_NOT_FOUND;
16398| }
16399| SetLastError(Err);
16400| return (ULONGLONG)-1;
16401| }
16402| BOOL DirWalkToEntryNum( tVolumeInfo *VolumeInfo,
16403| | PNTFS_File IAFile, ULONGLONG IASize, PINDEX_ALLOCATION
16404| | IA, PINDEX_ENTRY IE, int Level, ULONGLONG *Count,
16405| | ULONGLONG EntryNum, LPWIN32_FIND_DATA lpFindFileData )
16406| {
16407|     ULONG Err;
16408|     //DLOG((TEXT("%-*.s
16409|     | VCN=%016l64x\n"),Level*3,Level*3,TEXT("
16410|     | "),MyVCN));
16411|     while(1) {
16412|         if(IE->Flags & FLAGS_INDEX_SUBNODES) {
16413|             if(IAFile) {
16414|                 ULONGLONG MyVCN =
16415|                 | IA->VCNOfBuffer.QuadPart;
16416|                 ULONGLONG VCN =
16417|                 | NTFS_GetSubNodesVCN(IE);
16418|                 SAFE_ReadWrite(IA);
16419|                 Err = NTFS_ReadFile( IAFile,
16420|                 | VCN*NTFS_ClusterSizeInBytes(VolumeInfo), IASize, IA );
16421|                 _ASSERT(IA->AllocatedLength<IASize);
16422|                 if((Err==0) && (IA->Signature=='XDNI')
16423|                 | && (NTFS_FixupRecord( VolumeInfo, IA ))) {
16424|                     PINDEX_ENTRY NewIE;
16425|                     _ASSERT(IA->VCNOfBuffer.QuadPart
16426|                     | == VCN);
16427|                     SAFE_ReadOnly(IA);
16428|                     NewIE =
16429|                     | (PINDEX_ENTRY)&((BYTE*)IA)[0x18+IA->HeaderSize];
16430|                     if (DirWalkToEntryNum( VolumeInfo,
16431|                     | IAFile, IASize, IA, NewIE, Level+1,Count,EntryNum,

```

```

    | lpFindFileData )){
16422|         return (TRUE);
16423|     } // if not the root
16424|     if(MyVCN!=(ULONGLONG)-1) {
16425|         // read our data again
16426|         SAFE_ReadWrite(IA);
16427|         Err = NTFS_ReadFile( IFile,
    | MyVCN*NTFS_ClusterSizeInBytes(VolumeInfo), IASize, IA
    | );
16428|     | _ASSERTE(IA->AllocatedLength<IASize);
16429|         // already passed test above so
    | we dont need to check again...
16430|         NTFS_FixupRecord( VolumeInfo,
    | IA );
16431|         SAFE_ReadOnly(IA);
16432|     }
16433| } else {
16434|     SetLastError(ERROR_DISK_CORRUPT);
16435|     return (FALSE);
16436| }
16437| } else {
16438|     SetLastError(ERROR_DISK_CORRUPT);
16439|     return (FALSE);
16440| }
16441| }
16442|
16443| if(!(IE->Flags & FLAGS_INDEX_LAST)) {
16444|     if(IE->FileNameType != FILENAME_DOS) {
16445| #if 0
16446|         DLOG((TEXT("%04I64x: %012I64x
    | %-*.s\n"),
16447|             (*Count),
16448|             IE->DirectoryMft,
16449|             IE->FileNameLength,
16450|             IE->FileNameLength,
16451|             IE->FileName
16452|         ));
16453|     ##else
16454|         DLOG((
16455|             TEXT("%-*.s ")
16456|             TEXT("DirectoryMft=%012I64x,
    | SequenceNumber=%04I64x, ")
16457|             TEXT("EntrySize=%04x, DataSize=%04x
    | (%04x), Flags=%08x, ")
16458|             TEXT("MyDirectoryEntry=%012I64x,
    | MySequenceNumber=%04I64x, ")
16459|             | TEXT("AllocatedAttributeSize=%016I64x,
    | AttributeSize=%016I64x, ")

```

```

16460|         TEXT("Attributes=%016l64x (%s), ")
16461|         TEXT("%-*.*s\n"),
16462|         Level*3,Level*3,TEXT("
| "),
16463|         IE->DirectoryMft,
| IE->SequenceNumber, IE->EntrySize,
16464|
| IE->DataSize,IE->EntrySize-IE->DataSize, IE->Flags,
16465|         IE->MyDirectoryEntry,
| IE->MySequenceNumber,
16466|         IE->AllocatedAttributeSize,
| IE->AttributeSize, IE->Attributes,
| GetDosAttributes(IE->Attributes.LowPart),
16467|         IE->FileNameLength,
16468|         IE->FileNameLength,
16469|         IE->FileName
16470|     ));
16471| #endif
16472|     }
16473|     if (*Count==EntryNum) {
16474|         wcsncpy
| (lpFindFileData->cFileName,IE->FileName,IE->FileNameLeng
| th );
16475|
| lpFindFileData->cFileName[IE->FileNameLength] = 0;
16476|
16477|         return TRUE;
16478|     }
16479|     (*Count)++;
16480|
16481|     } else
16482|         break;
16483|
16484|     ((BYTE*)IE)+=NTFS_IndexEntrySize(IE);
16485| }
16486| return FALSE;
16487| }
16488|
16489| BOOL FindEntryNumInDir( tVolumeInfo *VolumeInfo,
| ULONGLONG MftEntry, ULONGLONG EntryNum,
| LPWIN32_FIND_DATA lpFindFileData )
16490| {
16491|     BOOL Found;
16492|     PNTFS_File IRFile=NTFS_OpenFileByNumber(
| VolumeInfo, MftEntry, INDEX_ROOT_ATTR, NULL );
16493|     ULONG Err=0;
16494|     if(IRFile) {
16495|         PNTFS_File IFile = NTFS_OpenFileByNumber(
| VolumeInfo, MftEntry, INDEX_ALLOCATION_ATTR, NULL );
16496|         ULONGLONG IRSize = NTFS_GetFileSize(IRFile);

```



```

16497|     PINDEX_ROOT IR    =
    | SAFE_Alloc((ULONG)IRSize);
16498|
16499|     //DumpMft(VolumeInfo,IAFile->Mft);
16500|     if(IR) {
16501|         PINDEX_ENTRY    IE =
    | (PINDEX_ENTRY)(IR+1);
16502|         Err = NTFS_ReadFile( IRFile, 0, IRSize, IR
    | );
16503|         SAFE_ReadOnly(IR);
16504|
16505|         if(!Err) {
16506|             ULONGLONG    IASize;
16507|             PINDEX_ALLOCATION IA=NULL;
16508|             ULONGLONG    Count=0;
16509|
16510|             if(IAFile) {
16511|                 IASize = IR->SizeOfIndex;
16512|                 IA    =
    | SAFE_Alloc((ULONG)IASize);
16513|                 if(IA) {
16514|                     IA->VCNOOfBuffer.QuadPart =
    | (ULONGLONG)-1;
16515|                     SAFE_ReadOnly(IA);
16516|                 } else {
16517|                     Err = ERROR_OUTOFMEMORY;
16518|                 }
16519|             }
16520|
16521|             if (((IAFile!=NULL) && (IA!=NULL)) ||
16522|                 ((IAFile==NULL))) {
16523|                 Found =
    | (DirWalkToEntryNum(VolumeInfo, IAFile, IASize, IA, IE,
    | 0,&Count, EntryNum, lpFindFileData ));
16524|                 DLOG((TEXT("Total files =
    | %l64d\n"),Count));
16525| #if 0
16526|                 DLOG((TEXT("%04l64x: %012l64x
    | %-*.s\n"),
16527|                     (Count),
16528|                     IE->DirectoryMft,
16529|                     IE->FileNameLength,
16530|                     IE->FileNameLength,
16531|                     IE->FileName
16532|                     ));
16533| #endif
16534|             }
16535|
16536|             if(IA)
16537|                 SAFE_Free(IA);

```

```

16538|         } else {
16539|             DLOG((TEXT("Error %08x reading from
| Index root\n"),Err));
16540|         }
16541|         SAFE_Free(IR);
16542|     } else {
16543|         DLOG((TEXT("Error, out of memory\n")));
16544|         Err = ERROR_OUTOFMEMORY;
16545|     }
16546|     if(!IAFile)
16547|         NTFS_CloseFile(IAFile);
16548|     NTFS_CloseFile(IRFile);
16549| } else {
16550|     // user passed in a filename as a path, example
| : \winnt\system32\calc.exe\file
16551|     DLOG((TEXT("Not a directory\n")));
16552|     Err = ERROR_PATH_NOT_FOUND;
16553| }
16554| SetLastError(Err);
16555| return Found;
16556| }
16557|
16558| BOOL StepToNextIndex( PNTFS_FIND_INFO FI, tVolumeInfo
| *VolumeInfo, LPWIN32_FIND_DATA lpFindFileData )
16559| {
16560|     ULONG Err;
16561|
16562|     if (1){ /*(!SorryShowsAllOverFolks) {
16563|
16564| //DLOG((TEXT("%-*.s
| VCN=%016l64x\n"),Level*3,Level*3,TEXT("
| "),MyVCN));
16565|
16566| // //We start by following wherever the tree leads
| passing any DOS names on the way
16567| // while((FI->DeepestNode->IE->Flags &
| FLAGS_INDEX_SUBNODES) || ((FI->DeepestNode->IE->Flags &
| FLAGS_INDEX_LAST)) ||
| (FI->DeepestNode->IE->FileNameType == FILENAME_DOS)) {
16568|         PNODE NodeElect;
16569|         while(FI->DeepestNode->IE->Flags &
| FLAGS_INDEX_SUBNODES) {
16570|
16571|             if(FI->IAFile) {
16572|                 ULONGLONG MyVCN =
| FI->DeepestNode->IA->VCNOfBuffer.QuadPart;
16573|                 ULONGLONG VCN =
| NTFS_GetSubNodesVCN(FI->DeepestNode->IE);
16574|
16575|                 NodeElect=

```

```

    | SAFE_Alloc((ULONG)(sizeof(tNODE)));
16576|         if(NodeElect) {
16577|             PINDEX_ALLOCATION NewIA =
    | SAFE_Alloc((ULONG)FI->IR->SizeOfIndex);
16578|             if(NewIA) {
16579|                 // what's this all about
16580| //not needed anymore??    IA->VCNOfBuffer.QuadPart =
    | (ULONGLONG)-1;
16581|             Err = NTFS_ReadFile(
    | FI->IAFile, VCN*NTFS_ClusterSizeInBytes(VolumeInfo),
    | FI->IR->SizeOfIndex, NewIA );
16582|
    | _ASSERT(NewIA->AllocatedLength<FI->IR->SizeOfIndex);
16583|             if((Err==0) &&
    | (NewIA->Signature=='XDNI') && (NTFS_FixupRecord(
    | VolumeInfo, NewIA ))) {
16584| //NNAM??             PINDEX_ENTRY NewIE;
16585|
    | _ASSERT(NewIA->VCNOfBuffer.QuadPart == VCN);
16586|             SAFE_ReadOnly(NewIA);
16587|             //Now we've built a new
    | node, link it in
16588|
    | NodeElect->ShallowerNode = FI->DeepestNode;
16589|             NodeElect->IA = NewIA;
16590|             NodeElect->IE =
    | (PINDEX_ENTRY)&((BYTE*)NewIA)[0x18+NewIA->HeaderSize];
16591|             FI->DeepestNode =
    | NodeElect;
16592|
16593|             /*         if (DirWalkToEntryNum(
    | VolumeInfo, IAFile, IASize, IA, NewIE,
    | Level+1,Count,EntryNum, lpFindFileData )){
16594|                 return (TRUE);
16595|             }// if not the root
16596|
    | if(MyVCN!=(ULONGLONG)-1) {
16597|                 // read our data
    | again
16598|                 SAFE_ReadWrite(IA);
16599|                 Err =
    | NTFS_ReadFile( IAFile,
    | MyVCN*NTFS_ClusterSizeInBytes(VolumeInfo), IASize, IA
    | );
16600|
    | _ASSERT(IA->AllocatedLength<IASize);
16601|                 // already passed
    | test above so we dont need to check again...
16602|                 NTFS_FixupRecord(
    | VolumeInfo, IA );

```

```

16603|             SAFE_ReadOnly(IA);
16604|         }
16605|    */
16606|        } else {
16607|
16608|            | SetLastError(ERROR_DISK_CORRUPT);
16609|            return (FALSE);
16610|        }
16611|        // Gotta make sure all
16612|        | these errors get all the way back
16613|    } else {
16614|        Err = ERROR_OUTOFMEMORY;
16615|    }
16616|    } else {
16617|        Err = ERROR_OUTOFMEMORY;
16618|    }
16619|    } else {
16620|        SetLastError(ERROR_DISK_CORRUPT);
16621|        return (FALSE);
16622|    }
16623|    }
16624|    while ((FI->DeepestNode->IE->Flags &
16625|        | FLAGS_INDEX_LAST)) {
16626|        //Delink and release deepest node here
16627|        if ((BYTE*)FI->DeepestNode ==
16628|            | (BYTE*)&FI->DeepestNode) {
16629|            SetLastError(ERROR_NO_MORE_FILES);
16630|            return (FALSE);
16631|        }
16632|        SAFE_Free(FI->DeepestNode->IA);
16633|        NodeElect =
16634|            | FI->DeepestNode->ShallowerNode;
16635|        SAFE_Free(FI->DeepestNode);
16636|        FI->DeepestNode = NodeElect;
16637|    }
16638|    } //    } //We'll only get to here when we've followed
16639|    | got through the tree to a valid file
16640|
16641|
16642|
16643|
16644|    } else {
16645|        return FALSE;
16646|    }
16647|    }
16648|    return TRUE;
16649|}
16650|
16651|
16652|
16653|
16654|
16655|
16656|
16657|
16658|
16659|
16660|
16661|
16662|
16663|
16664|
16665|
16666|
16667|
16668|
16669|
16670|
16671|
16672|
16673|
16674|
16675|
16676|
16677|
16678|
16679|
16680|
16681|
16682|
16683|
16684|
16685|
16686|
16687|
16688|
16689|
16690|
16691|
16692|
16693|
16694|
16695|
16696|
16697|
16698|
16699|
16700|
16701|
16702|
16703|
16704|
16705|
16706|
16707|
16708|
16709|
16710|
16711|
16712|
16713|
16714|
16715|
16716|
16717|
16718|
16719|
16720|
16721|
16722|
16723|
16724|
16725|
16726|
16727|
16728|
16729|
16730|
16731|
16732|
16733|
16734|
16735|
16736|
16737|
16738|
16739|
16740|
16741|
16742|
16743|
16744|
16745|
16746|
16747|
16748|
16749|
16750|
16751|
16752|
16753|
16754|
16755|
16756|
16757|
16758|
16759|
16760|
16761|
16762|
16763|
16764|
16765|
16766|
16767|
16768|
16769|
16770|
16771|
16772|
16773|
16774|
16775|
16776|
16777|
16778|
16779|
16780|
16781|
16782|
16783|
16784|
16785|
16786|
16787|
16788|
16789|
16790|
16791|
16792|
16793|
16794|
16795|
16796|
16797|
16798|
16799|
16800|
16801|
16802|
16803|
16804|
16805|
16806|
16807|
16808|
16809|
16810|
16811|
16812|
16813|
16814|
16815|
16816|
16817|
16818|
16819|
16820|
16821|
16822|
16823|
16824|
16825|
16826|
16827|
16828|
16829|
16830|
16831|
16832|
16833|
16834|
16835|
16836|
16837|
16838|
16839|
16840|
16841|
16842|
16843|
16844|
16845|
16846|
16847|
16848|
16849|
16850|
16851|
16852|
16853|
16854|
16855|
16856|
16857|
16858|
16859|
16860|
16861|
16862|
16863|
16864|
16865|
16866|
16867|
16868|
16869|
16870|
16871|
16872|
16873|
16874|
16875|
16876|
16877|
16878|
16879|
16880|
16881|
16882|
16883|
16884|
16885|
16886|
16887|
16888|
16889|
16890|
16891|
16892|
16893|
16894|
16895|
16896|
16897|
16898|
16899|
16900|
16901|
16902|
16903|
16904|
16905|
16906|
16907|
16908|
16909|
16910|
16911|
16912|
16913|
16914|
16915|
16916|
16917|
16918|
16919|
16920|
16921|
16922|
16923|
16924|
16925|
16926|
16927|
16928|
16929|
16930|
16931|
16932|
16933|
16934|
16935|
16936|
16937|
16938|
16939|
16940|
16941|
16942|
16943|
16944|
16945|
16946|
16947|
16948|
16949|
16950|
16951|
16952|
16953|
16954|
16955|
16956|
16957|
16958|
16959|
16960|
16961|
16962|
16963|
16964|
16965|
16966|
16967|
16968|
16969|
16970|
16971|
16972|
16973|
16974|
16975|
16976|
16977|
16978|
16979|
16980|
16981|
16982|
16983|
16984|
16985|
16986|
16987|
16988|
16989|
16990|
16991|
16992|
16993|
16994|
16995|
16996|
16997|
16998|
16999|
17000|

```

```

16647| #if 0
16648|         DLOG((TEXT("%04l64x: %012l64x
| %-*.s\n"),
16649|             (*Count),
16650|             IE->DirectoryMft,
16651|             IE->FileNameLength,
16652|             IE->FileNameLength,
16653|             IE->FileName
16654|         ))
16655| //else
16656|         DLOG((
16657|             TEXT("%-*.s ")
16658|             TEXT("DirectoryMft=%012l64x,
| SequenceNumber=%04l64x, ")
16659|             TEXT("EntrySize=%04x, DataSize=%04x
| (%04x), Flags=%08x, ")
16660|             TEXT("MyDirectoryEntry=%012l64x,
| MySequenceNumber=%04l64x, ")
16661|             | TEXT("AllocatedAttributeSize=%016l64x,
| AttributeSize=%016l64x, ")
16662|             TEXT("Attributes=%016l64x (%s), ")
16663|             TEXT("%-*.s\n"),
16664|             Level*3,Level*3,TEXT("
| "),
16665|             IE->DirectoryMft,
| IE->SequenceNumber, IE->EntrySize,
16666|             | IE->DataSize,IE->EntrySize-IE->DataSize, IE->Flags,
16667|             IE->MyDirectoryEntry,
| IE->MySequenceNumber,
16668|             IE->AllocatedAttributeSize,
| IE->AttributeSize, IE->Attributes,
| GetDosAttributes(IE->Attributes.LowPart),
16669|             IE->FileNameLength,
16670|             IE->FileNameLength,
16671|             IE->FileName
16672|         ))
16673| #endif
16674|         ;
16675|
16676| */
16677|
16678| BOOL OpenIndexRoot( PNTFS_FIND_INFO FI, tVolumeInfo
| *VolumeInfo, ULONGLONG MftEntry, LPWIN32_FIND_DATA
| lpFindFileData )
16679| {
16680| // BOOL Found;
16681| ULONG Err=0;
16682| FI->IRFile=NTFS_OpenFileByNumber( VolumeInfo,

```

```

    | MftEntry, INDEX_ROOT_ATTR, NULL );
16683|   if(FI->IRFile) {
16684|       FI->IAFile = NTFS_OpenFileByNumber(
    | VolumeInfo, MftEntry, INDEX_ALLOCATION_ATTR, NULL );
16685|       FI->IRSize = NTFS_GetFileSize(FI->IRFile);
16686|       FI->IR     = SAFE_Alloc((ULONG)FI->IRSize);
16687|
16688|       //DumpMft(VolumeInfo,IAFile->Mft);
16689|       if(FI->IR) {
16690|           FI->IE = (PINDEX_ENTRY)(FI->IR+1);
16691|           Err = NTFS_ReadFile( FI->IRFile, 0,
    | FI->IRSize, FI->IR );
16692|           SAFE_ReadOnly(FI->IR);
16693|
16694|           if(!Err) {
16695|
    | FI->DeepestNode=(PNODE)&FI->DeepestNode;
16696|           return TRUE;
16697|
16698| /*
16699|         ULONGLONG      Count=0;
16700|
16701|         if(FI->IAFile) {
16702|             FI->IASize = FI->IR->SizeOfIndex;
16703|             FI->IA      =
    | SAFE_Alloc((ULONG)FI->IASize);
16704|             if(FI->IA) {
16705|                 FI->IA->VCNOOfBuffer.QuadPart =
    | (ULONGLONG)-1;
16706|                 SAFE_ReadOnly(IA);
16707|             } else {
16708|                 Err = ERROR_OUTOFMEMORY;
16709|             }
16710|         }
16711|
16712|         if (((IAFile!=NULL) && (IA!=NULL)) ||
16713|             ((IAFile==NULL))) {
16714|             Found =
    | (DirWalkToEntryNum(VolumeInfo, IAFile, IASize, IA, IE,
    | 0,&Count, EntryNum, lpFindFileData ));
16715|             DLOG((TEXT("Total files =
    | %!64d\n"),Count));
16716| #if 0
16717|             DLOG((TEXT("%04!64x: %012!64x
    | %-*. *s\n"),
16718|                 (Count),
16719|                 IE->DirectoryMft,
16720|                 IE->FileNameLength,
16721|                 IE->FileNameLength,
16722|                 IE->FileName

```

```

16723|         ));
16724| #endif
16725|     }
16726|     if(IA)
16727|         SAFE_Free(IA);
16728| /*
16729|     } else {
16730|         DLOG((TEXT("Error %08x reading from
| Index root\n"),Err));
16731|     }
16732|     SAFE_Free(FI->IR);
16733| } else {
16734|     DLOG((TEXT("Error, out of memory\n")));
16735|     Err = ERROR_OUTOFMEMORY;
16736| }
16737| if(FI->IAFile)
16738|     NTFS_CloseFile(FI->IAFile);
16739|     NTFS_CloseFile(FI->IRFile);
16740| } else {
16741|     // user passed in a filename as a path, example
| : \winnt\system32\calc.exe\file
16742|     DLOG((TEXT("Not a directory\n")));
16743|     Err = ERROR_PATH_NOT_FOUND;
16744| }
16745| SetLastError(Err);
16746| return FALSE;
16747| }
16748|
16749| /*
16750| Opens file with its name. Type is the
| attribute(stream) to open.
16751| Must be a full path, not a relative path.
16752| */
16753| PNTFS_File NTFS_OpenFileByName ( pVolumeInfo
| VolumeInfo, WCHAR *Name, ULONG Type, WCHAR *AttrName )
16754| {
16755|     ULONGLONG MftEntry;
16756|     // we only can handle full paths
16757|     if(*Name==L'\\') {
16758|         Name++;
16759|
16760|         if(*Name=='\0') {
16761|             // special case for the root (Name passed
| in was '\')
16762|             // to open the Name way, open '\.'
16763|             return NTFS_OpenFileByNumber( VolumeInfo,
| FILE_ROOT, Type, AttrName );
16764|         } else {
16765|             // look for the entry from the root
| directory

```

```

16766|         MftEntry = NTFS_RecurseDir( VolumeInfo,
    | FILE_ROOT, Name );
16767|         if(MftEntry!=(ULONGLONG)-1) {
16768|             return NTFS_OpenFileByNumber(
    | VolumeInfo, MftEntry, Type, AttrName );
16769|         } else {
16770|             DLOG((TEXT("Error %08x opening
    | file\n"),GetLastError()));
16771|         }
16772|     }
16773| }
16774| return NULL;
16775| }
16776|
16777| /*
16778|  Closes the file and frees memory associated with it
16779| */
16780| ULONG NTFS_CloseFile( PNTFS_File File )
16781| {
16782|     if(File) {
16783|         if(File->Mft)
16784|             SAFE_Free(File->Mft);
16785|         if(File->VCNToLCNMappingTable)
16786|             SAFE_Free(File->VCNToLCNMappingTable);
16787|         if(File->Buffer)
16788|             SAFE_Free(File->Buffer);
16789|         if(File->CompressedBuffer)
16790|             SAFE_Free(File->CompressedBuffer);
16791|         if(File->UnCompressedBuffer)
16792|             SAFE_Free(File->UnCompressedBuffer);
16793|
16794|         SAFE_Free(File);
16795|     }
16796|     return 0;
16797| }
16798|
16799| ULONG NTFS_ReadCompressedCluster( PNTFS_File File,
16800|                                     ULONGLONG VCN,
16801|                                     ULONGLONG
    | *VCNInMemory,
16802|                                     PCHAR
    | CompressedBuffer,
16803|                                     PCHAR
    | UnCompressedBuffer )
16804| {
16805|     ULONGLONG LCN;
16806|     ULONG Count=0;
16807|     ULONG Err=0;
16808|     ULONG CS=NTFS_ClusterSizeInBytes(File->VolumeInfo);
16809|     //#define CS NTFS_ClusterSizeInBytes(File->VolumeInfo)

```



```

16810|
16811|  (*VCNInMemory) = (VCN / 16) * 16;    // rounded
      | down to nearest 16 clusters
16812|
16813|  for(Count=0;Count<16;Count++) {
16814|      LCN = NTFS_VCNToLCN( File, (*VCNInMemory)+Count
      | );
16815|      if(LCN!=(ULONGLONG)-1) {
16816|          Err = NTFS_ReadLogicalCluster(
      | File->VolumeInfo,
16817|                                LCN,
16818|                                1,
16819|
      | &CompressedBuffer[Count*CS]);
16820|      } else {
16821|          // sparse file area (which would only occur
      | if the file was
16822|          // compressed
16823|          memset(&CompressedBuffer[Count*CS],0,CS);
16824|          Err = 0;
16825|      }
16826|      if(Err) break;
16827|  }
16828|  if(!Err)
16829|      | NTFS-DecompressBuffer(UnCompressedBuffer,CompressedBuffere
      | r,16*CS);
16830|
16831|  return Err;
16832| }
16833|
16834| /*
16835|  Gets the size of the attribute(stream) specified in
      | the open
16836|  Note: This is NOT the same as the file size of the
      | file returned in a dir. To get
16837|  that information the File needs to be opened with
      | the ATTR_DATA stream or retrieved
16838|  from the FILENAME_ATTR or STANDARD_INFORMATION_ATTR
16839|  */
16840| ULONGLONG NTFS_GetFileSize ( PNTFS_File File )
16841| {
16842|  PATTRIBUTE Attribute = NTFS_GetAttributeByName(
      | File->VolumeInfo, File->Mft, File->Type, File->AttrName
      | );
16843|  if(Attribute) {
16844|      //DLOG((TEXT("File Size =
      | %!64d\n"),NTFS_GetDataSize(Attribute)));
16845|      return NTFS_GetDataSize(Attribute);
16846|  } else {

```

```

16847|     DLOG((TEXT("Error getting attribute\n")));
16848|     SetLastError(ERROR_NO_ATTRIBUTE_FOUND);
16849| }
16850| return (ULONGLONG)-1;
16851| }
16852|
16853| /*
16854| This Writes the data of the Attribute(Stream)
| specified in the open
16855| */
16856| ULONG NTFS_WriteFile ( PNTFS_File File,
16857|                        ULONGLONG ByteOffset,
16858|                        ULONGLONG ByteCount,
16859|                        PVOID Data )
16860| {
16861|     PATTRIBUTE
| Attribute=NTFS_GetAttributeByName(File->VolumeInfo,File-
| >Mft,File->Type,File->AttrName);
16862|     ULONG Err;
16863|
16864|     if(Attribute) {
16865|
16866|         if(Attribute->Compressed) {
16867|             DLOG((TEXT("Compression not supported
| now!\n")));
16868|             return ERROR_NOT_SUPPORTED;
16869|         }
16870|
16871|         if(Attribute->NotResident) {
16872|             // data is on disk
16873|             ULONGLONG StartVCN    = ByteOffset /
| (NTFS_ClusterSizeInBytes(File->VolumeInfo));
16874|             ULONGLONG StartOffset = ByteOffset %
| (NTFS_ClusterSizeInBytes(File->VolumeInfo));
16875|             ULONGLONG LastVCN    =
| (ByteOffset+ByteCount) /
| (NTFS_ClusterSizeInBytes(File->VolumeInfo));
16876|             ULONGLONG LastOffset =
| (ByteOffset+ByteCount) %
| (NTFS_ClusterSizeInBytes(File->VolumeInfo));
16877|             ULONGLONG VCN;        // offset into file
16878|             ULONGLONG LCN;        // offset into
| volume
16879|             ULONGLONG Count      = LastVCN +
| (LastOffset ? 1 : 0);
16880|             ULONGLONG VCNInMemory = (ULONGLONG)-1;
16881|
16882|             SAFE_ReadWrite( File->Buffer );
16883|             for(VCN=StartVCN;VCN<Count;VCN++) {
16884|                 // how we are going to handle this, is

```

```

| we will always read a cluster into memory
16885|          // modify the inmemory contents, and
| write it out again. This is slow, but it will
16886|          // work for our needs and simplifies
| the code needed to do the writes.
16887|
16888|          LCN = NTFS_VCNTToLCN( File, VCN );
16889|
16890|          if(LCN!=(ULONGLONG)-1) {
16891|              // read the cluster into memory
16892|              Err = NTFS_ReadLogicalCluster(
| File->VolumeInfo,
16893|
| LCN,
16894|              1,
16895|
| File->Buffer );
16896|          } else {
16897|              // sparse file area (which would
| only occur if the file was
16898|              // compressed
16899|              // we dont support writing to
| sparse areas as that would require
16900|              // code to extend the file
| allocation maps
16901|              Err = ERROR_NOT_SUPPORTED;
16902|          }
16903|
16904|          if(!Err) {
16905|              // now copy passed in data into
| double buffer
16906|              if(VCN==StartVCN) {
16907|                  if(VCN==LastVCN) {
16908|                      // only 1 or less cluster
| being copied
16909|                      // TODO: length is an int
| not an __int64
16910|
| memcpy(&((BYTE*)File->Buffer)[StartOffset],Data,(unsigne
| d int)(LastOffset-StartOffset));
16911|                      (BYTE*)Data+=LastOffset;
16912|                  } else {
16913|                      // VCN is first
16914|
| memcpy(&((BYTE*)File->Buffer)[StartOffset],Data,(unsigne
| d
| int)((NTFS_ClusterSizeInBytes(File->VolumeInfo))-StartOf
| fset));
16915|
| (BYTE*)Data+=NTFS_ClusterSizeInBytes(File->VolumeInfo);

```

```

16916|         }
16917|     } else
16918|     {
16919|         if(VCN==LastVCN) {
16920|             // on last VCN
16921|             memcpy(File->Buffer,Data,(unsigned int)LastOffset);
16922|             (BYTE*)Data+=LastOffset;
16923|         } else {
16924|             // VCN's in the middle
16925|             memcpy(File->Buffer,Data,(unsigned
16926|             | int)NTFS_ClusterSizeInBytes(File->VolumeInfo));
16927|             (BYTE*)Data+=NTFS_ClusterSizeInBytes(File->VolumeInfo);
16928|         }
16929|         // now write cluster to disk
16930|         Err = NTFS_WriteLogicalCluster(
16931|             | File->VolumeInfo,
16932|             | LCN,
16933|             | 1,
16934|             | File->Buffer );
16935|         if(Err) {
16936|             DLOG((TEXT("Error %08x writing
16937|             | LCN %l64d VCN %l64d\n"),Err,LCN,VCN));
16938|         }
16939|     } else {
16940|         DLOG((TEXT("Error %08x reading LCN
16941|         | %l64d VCN %l64d\n"),Err,LCN,VCN));
16942|         break;
16943|     }
16944|     SAFE_ReadOnly( File->Buffer );
16945| } else {
16946|     // okay, we are now working on resident mft
16947|     | attributes
16948|     // we will only replace the in memory
16949|     | copies, until a NTFS_CommitFile is called
16950|     if(NTFS_GetDataSize(Attribute)==ByteCount)
16951|     {
16952|         // okay, do a replacement
16953|         memcpy(Attribute,Data,(unsigned
16954|         | long)ByteCount);
16955|         Err = 0;
16956|     } else {
16957|         DLOG((TEXT("Byte count doesnt match
16958|         | last attribute size\n")));

```

```

16952|         Err = ERROR_INVALID_PARAMETER;
16953|     }
16954| }
16955| } else {
16956|     DLOG((TEXT("Attribute %08x could not be
| found\n"),File->Type));
16957|     Err = ERROR_NO_ATTRIBUTE_FOUND;
16958| }
16959| return Err;
16960| }
16961|
16962|
16963| /*
16964| This commits the data of the Attribute(Stream)
| specified in the open to disk
16965| */
16966| ULONG NTFS_CommitFile ( PNTFS_File File )
16967| {
16968|     return NTFS_WriteMftEntryNumber( File->VolumeInfo,
| File->MftEntryNum, File->Mft);
16969| }
16970|
16971| /*
16972| This reads the data of the Attribute(Stream)
| specified in the open
16973| */
16974| ULONG NTFS_ReadFile ( PNTFS_File File,
16975|         ULONGLONG ByteOffset,
16976|         ULONGLONG ByteCount,
16977|         PVOID Data )
16978| {
16979|     PATTRIBUTE
| Attribute=NTFS_GetAttributeByName(File->VolumeInfo,File-
| >Mft,File->Type,File->AttrName);
16980|     ULONG Err;
16981|
16982|     if(Attribute) {
16983|
16984|         if(Attribute->Compressed) {
16985|
| if(Attribute->NonResidentData.CompressionEngine!=4) {
16986|             DLOG((TEXT("Unsupported compression
| engine!!\n")));
16987|             return ERROR_NOT_SUPPORTED;
16988|         }
16989|     }
16990|
16991|     if(Attribute->NotResident) {
16992|         // data is on disk
16993|         ULONGLONG StartVCN    = ByteOffset /

```

```

    | (NTFS_ClusterSizeInBytes(File->VolumeInfo));
16994|     ULONGLONG StartOffset = ByteOffset %
    | (NTFS_ClusterSizeInBytes(File->VolumeInfo));
16995|     ULONGLONG LastVCN =
    | (ByteOffset+ByteCount) /
    | (NTFS_ClusterSizeInBytes(File->VolumeInfo));
16996|     ULONGLONG LastOffset =
    | (ByteOffset+ByteCount) %
    | (NTFS_ClusterSizeInBytes(File->VolumeInfo));
16997|     ULONGLONG VCN; // offset into file
16998|     ULONGLONG LCN; // offset into
    | volume
16999|     ULONGLONG Count = LastVCN +
    | (LastOffset ? 1 : 0);
17000|     ULONGLONG VCNInMemory = (ULONGLONG)-1;
17001|
17002|     SAFE_ReadWrite( File->Buffer );
17003|     for(VCN=StartVCN;VCN<Count;VCN++) {
17004|         // read the cluster into memory
17005|         if(Attribute->Compressed) {
17006|             if((VCN < VCNInMemory) || (VCN >=
    | VCNInMemory+16)) {
17007|                 // time to read compressed data
    | in
17008|                 Err =
    | NTFS_ReadCompressedCluster( File,
17009|                 | VCN,
17010|                 | &VCNInMemory,
17011|                 | File->CompressedBuffer,
17012|                 | File->UnCompressedBuffer
17013|                 | );
17014|             } else {
17015|                 // already in memory
17016|             }
17017|
    | memmove(File->Buffer,&File->UnCompressedBuffer[(VCN-VCNInMemory)*NTFS_ClusterSizeInBytes(File->VolumeInfo)],NTFS
    | _ClusterSizeInBytes(File->VolumeInfo));
17018|         } else {
17019|             LCN = NTFS_VCNToLCN( File, VCN );
17020|
17021|             if(LCN!=(ULONGLONG)-1) {
17022|                 Err = NTFS_ReadLogicalCluster(
    | File->VolumeInfo,
17023|

```

```

    | LCN,
17024|
    | 1,
17025|
    | File->Buffer );
17026|         } else {
17027|             // sparse file area (which
    | would only occur if the file was
17028|             // compressed
17029|
    | memset(File->Buffer,0,NTFS_ClusterSizeInBytes(File->Volu
    | meInfo));
17030|             Err = 0;
17031|         }
17032|     }
17033|
17034|     if(!Err) {
17035|         // now copy into user buffer
17036|         if(VCN==StartVCN) {
17037|             if(VCN==LastVCN) {
17038|                 // only 1 or less cluster
    | being copied
17039|                 // TODO: length is an int
    | not an __int64
17040|
    | memcpy(Data,&((BYTE*)File->Buffer)[StartOffset],(unsigne
    | d int)(LastOffset-StartOffset));
17041|                 (BYTE*)Data+=LastOffset;
17042|             } else {
17043|                 // VCN is first
17044|
    | memcpy(Data,&((BYTE*)File->Buffer)[StartOffset],(unsigne
    | d
    | int)((NTFS_ClusterSizeInBytes(File->VolumeInfo))-StartOf
    | fset));
17045|
    | (BYTE*)Data+=NTFS_ClusterSizeInBytes(File->VolumeInfo);
17046|             }
17047|         } else
17048|         if(VCN==LastVCN) {
17049|             // on last VCN
17050|
    | memcpy(Data,File->Buffer,(unsigned int)LastOffset);
17051|             (BYTE*)Data+=LastOffset;
17052|         } else {
17053|             // VCN's in the middle
17054|
    | memcpy(Data,File->Buffer,(unsigned
    | int)NTFS_ClusterSizeInBytes(File->VolumeInfo));
17055|

```

```

    | (BYTE*)Data+=NTFS_ClusterSizeInBytes(File->VolumeInfo);
17056|         }
17057|     } else {
17058|         DLOG((TEXT("Error %08x reading LCN
    | %l64d VCN %l64d\n"),Err,LCN,VCN));
17059|         break;
17060|     }
17061| }
17062|     SAFE_ReadOnly( File->Buffer );
17063| } else {
17064|     // data is in mft
17065|     BYTE
    | *AttrData=NTFS_GetDataPointer(Attribute);
17066|     if(AttrData) {
17067|         // TODO: length is an int, not an
    | __int64
17068|         | memcpy(Data,&AttrData[ByteOffset],min((unsigned
    | int)ByteCount,Attribute->ResidentData.DataLength));
17069|         Err = NO_ERROR;
17070|     } else {
17071|         DLOG((TEXT("Data for attribute not
    | found!!!\n")));
17072|         Err = ERROR_NO_ATTRIBUTE_FOUND;
17073|     }
17074| }
17075| } else {
17076|     DLOG((TEXT("Attribute %08x could not be
    | found\n"),File->Type));
17077|     Err = ERROR_NO_ATTRIBUTE_FOUND;
17078| }
17079| return Err;
17080| }
17081|
17082| /*
17083|  returns the filename, file times, and sizes of the
    | file.
17084|  To minimize access to the disk, this routine gets
    | the attribute data from the current
17085|  mft that was opened, instead of opening the file
    | for the FILENAME_ATTR. This can be
17086|  done because FILENAME_ATTR attributes are always
    | resident and in the mft. But incase
17087|  i am wrong, that case is checked also.. ;)
17088| */
17089| ULONG NTFS_GetFileInfo( PNTFS_File File, PFILENAME FN )
17090| {
17091|     PNTFS_File FNFile;
17092|     ULONG Err=0;
17093|     PATTRIBUTE Attribute=NTFS_GetAttributeByName(

```



```

    | File->VolumeInfo, File->Mft, FILENAME_ATTR, NULL );
17094|     if(Attribute) {
17095|         PVOID Data=NTFS_GetDataPointer(Attribute);
17096|         // if the attribute is on disk, read it
17097|         if(!Data) || (Attribute->NotResident))
17098|             goto DoFileWay;
17099|
17100|         // the attribute is in memory!
17101|         memcpy(FN,Data, sizeof(FILENAME));
17102|     } else {
17103|         // attribute not found or
17104| DoFileWay:
17105|         // attribute on disk
17106|         FNFile = NTFS_OpenFileByNumber (
            | File->VolumeInfo, File->MftEntryNum, FILENAME_ATTR,
            | NULL );
17107|
17108|         if(FNFile) {
17109|             Err = NTFS_ReadFile(FNFile, 0,
            | sizeof(FILENAME), FN );
17110|             if(Err) {
17111|                 DLOG((TEXT("Error %08x reading
            | file\n"),Err));
17112|             }
17113|             NTFS_CloseFile(FNFile);
17114|         } else {
17115|             DLOG((TEXT("No filename attribute
            | found\n"))));
17116|             Err = ERROR_NO_ATTRIBUTE_FOUND;
17117|         }
17118|     }
17119|     return Err;
17120| }
17121|
17122| /*
17123|  if a file is compressed returns its compressed
    | size, otherwise -1
17124|  the compression ratio = NTFS_GetCompressedFileSize
    | / NTFS_GetFileSize;
17125| */
17126| ULONGLONG NTFS_GetCompressedFileSize( PNTFS_File File )
17127| {
17128|     PATTRIBUTE Attribute = NTFS_GetAttributeByName(
    | File->VolumeInfo, File->Mft, File->Type, File->AttrName
    | );
17129|     if(Attribute) {
17130|         // only non resident data can be compressed
17131|         if((Attribute->Compressed) &&
            | (Attribute->NotResident)) {
17132|             DLOG((TEXT("Compressed File Size =

```

```

    | %l64d\n"),Attribute->NonResidentData.CompressedSize.Quad
    | Part));
17133|         return
    | Attribute->NonResidentData.CompressedSize.QuadPart;
17134|     } else {
17135|         // not compressed
17136|         SetLastError(ERROR_NOT_COMPRESSED);
17137|     }
17138| } else {
17139|     DLOG((TEXT("Error getting attribute\n")));
17140|     SetLastError(ERROR_NO_ATTRIBUTE_FOUND);
17141| }
17142| return (ULONGLONG)-1;
17143| }
17144|
17145| /*
17146| deal with data run being external
17147| */
17148| PVOID NTFS_GetDataRun(PATTRIBUTE Attribute, PNTFS_File
    | NtfsFile, pVolumeInfo VolumeInfo, ULONGLONG
    | MftEntryNum, ULONG Type, WCHAR *AttrName )
17149| {
17150|     ULONG Err ;
17151|     PATTRIBUTE BaseAttribute=Attribute;
17152|     PVOID DataRun = NTFS_GetDataPointer( Attribute );
17153|     // Addresss the Attributelist attribute which is
    | the control mechanism for nonresident
17154|     // attributes in which even the run lists are so
    | large they are also non resident.
17155|     PATTRIBUTE pAttributeList =
    | NTFS_GetAttributeByName( VolumeInfo, NtfsFile->Mft,
    | ATTRIBUTE_LIST_ATTR, NULL );
17156|     if (pAttributeList) {
17157|         // NOTE! All the following complication is to
    | enable collection...
17158|         // .....finish this thought.....and put
    | it in the right place.
17159|
    | //...-----
    | -----
17160|         // TODO!! WHAT if this AttributeList attribute
    | itself is non-resident.
17161|         // Our info resource says that's possible!! And
    | could reasonably likely
17162|         // happen for a very fragged file.
17163|         // Of course it had better not need to have an
    | entry for itself !!
17164|         //Start at 1st Attribute record
17165|         PVOID DataRunComplex;
17166|         PVOID DataRunComplexLimit;

```

```

17167|     ULONG ThisDataRunPart ;
17168|     PATTRIBUTE_LIST pAttributeExtension =
| NTFS_GetDataPointer(pAttributeList) ;
17169|     ULONG CountRunExtensions = 0 ;
17170|     while (((BYTE*)pAttributeExtension <
| (BYTE*)pAttributeList + pAttributeList->Length) &&
| (pAttributeExtension->Type <= Type)){
17171|         // TODO What about names!! (Which brings
| up the question-
17172|         // is it the name of an individual
| attribute or an attribute type??
17173|         if (pAttributeExtension->Type == Type) {
17174|             CountRunExtensions++ ;
17175|         }
17176|         (BYTE*)pAttributeExtension +=
| sizeof(ATTRIBUTE_LIST) ;
17177|     }
17178|     if (CountRunExtensions) {
17179|         //TODO Tune this size!! Using simple max.
| of total blocks for now
17180|         DataRunComplexLimit = DataRunComplex =
| SAFE_Alloc(CountRunExtensions*(NTFS_MftByteSize(VolumeIn
| fo)));
17181|         if (DataRunComplex) {
17182|             PMFT ExtensionMft =
| SAFE_Alloc(NTFS_MftByteSize(VolumeInfo));
17183|             if (ExtensionMft) {
17184|                 //Restart at 1st Attribute record
17185|                 pAttributeExtension =
| NTFS_GetDataPointer(pAttributeList);
17186|                 while (((BYTE*)pAttributeExtension
| < (BYTE*)pAttributeList + pAttributeList->Length) &&
| (pAttributeExtension->Type <= Type)){
17187|                     // TODO What about names!!
| (Which brings up the question-
17188|                     // is it the name of an
| individual attribute or an attribute type??
17189|                     if (pAttributeExtension->Type
| == Type) {
17190|                         ULONGLONG TargetCluster ;
17191|                         if
| (pAttributeExtension->MFTEntry == 0) {
17192|                             TargetCluster=
| FindClusterForEntryNum( DataRun,
| pAttributeExtension->MFTEntry) ;
17193|                         } else {
17194|                             TargetCluster=
| FindClusterForEntryNum( DataRunComplex,
| pAttributeExtension->MFTEntry*VolumeInfo->NtfsBootSector
| ->MFTRecordSize) ;

```

```

17195|         }
17196|         Err =
| NTFS_ReadLogicalCluster(VolumeInfo,TargetCluster,VolumeI
| nfo->NtfsBootSector->MFTRecordSize, ExtensionMft) ;
17197| /*//             if (!Err) "was an
| opening curly baracket here!!!" */
17198|             //TODO Even more thorough
| checks that it belongs
17199|             if((Err==0) &&
| (ExtensionMft->Signature=='ELIF') && (NTFS_FixupRecord(
| VolumeInfo, ExtensionMft ))) {
17200|                 Attribute =
| NTFS_GetAttributeByName( VolumeInfo, ExtensionMft,
| Type, AttrName );
17201|                 if(Attribute) {
17202|                     //We've already
| established base record says non resident so all
17203|                     // extension
| records presum. say same - or else where are we??
17204|                     | _ASSERTE(Attribute->NotResident) ;
17205|                     // find its datarun
17206|                     DataRun =
| NTFS_GetDataPointer( Attribute );
17207|                     // find its
| compressed data run length
17208|                     ThisDataRunPart=
| GetPackedDataRunLength(
| DataRun,Attribute->NonResidentData.LastVCN.QuadPart-Attr
|  ibute->NonResidentData.StartingVCN.QuadPart+1 );
17209|                     // do I need an
| error chk??? he doesn't
17210|                     if
| (GetByte(DataRunComplex,(0)))
17211|                     | AdjustGapToRelative( DataRunComplex, DataRun,
| DataRunComplexLimit );
17212|                     | memcpy(DataRunComplexLimit,DataRun,ThisDataRunPart);
17213|                     (ULONG)
| DataRunComplexLimit += ThisDataRunPart ;
17214|                     //update original
| base mft's LastVCN to reflect extended complex. This is
| OK cos
17215|                     //we're only
| changing memory value in aread-only environment!!
17216|                     | BaseAttribute->NonResidentData.LastVCN.QuadPart =
| Attribute->NonResidentData.LastVCN.QuadPart;
17217|                     } else {

```

```

17218|             DLOG((TEXT("No %02x
| attribute found in file!\n"),Type));
17219|             | SetLastError(ERROR_NO_ATTRIBUTE_FOUND);
17220|             }
17221|             } else {
17222|             DLOG((TEXT("Error %08x
| reading Mft\n"),Err));
17223|             SetLastError(Err);
17224|             }
17225| //             need to release and give
| bad status
17226|             }
17227|             (BYTE*)pAttributeExtension +=
| sizeof(ATTRIBUTE_LIST) ;
17228|             }
17229|             SAFE_Free(ExtensionMft);
17230|             return (DataRunComplex) ;
17231|
17232|
17233|             } else {
17234|             DLOG((TEXT("Error out of
| memory\n")));
17235|             SetLastError(ERROR_OUTOFMEMORY);
17236|             }
17237|             } else {
17238|             DLOG((TEXT("Error out of memory\n")));
17239|             SetLastError(ERROR_OUTOFMEMORY);
17240|             }
17241|             return 0;
17242|
17243|             }
17244|     }
17245|     return (DataRun) ;
17246|     // need to point datarun -> dataruncomplex
17247| }
17248|
17249| /*
17250|     Fills in a array of the files logical clusters.
17251|     DataRun is the Data pointer of the non resident
| attribute.
17252| */
17253| ULONG MakeVCNTtoLCNMapping ( PVOID DataRun, PDATA_RUN
| MappingTable, ULONG NumRuns )
17254| {
17255|     ULARGE_INTEGER Cluster={0};
17256|     ULARGE_INTEGER Length={0};
17257|     ULONGLONG Run=0;
17258|     ULONG Offset=0;
17259|     ULONG Sparse=0;

```

```

17260|
17261|   while(Run<NumRuns) {
17262|
17263|       | if(GetRun(DataRun,&Offset,&Cluster,&Length,&Sparse)) {
17264|           //DLOG((TEXT("Offset = %12d,
17265|           | Cluster=%12l64d, Length=%12l64d,
17266|           | Sparse=%d\n"),Offset,Cluster, Length,Sparse));
17267|           if(!Sparse)
17268|               MappingTable[Run].Cluster =
17269|               | Cluster.QuadPart;
17270|           else
17271|               MappingTable[Run].Cluster =
17272|               | (ULONGLONG)-1;
17273|           MappingTable[Run++].Length =
17274|           | Length.QuadPart;
17275|       } else {
17276|           return ERROR_INVALID_RUN;
17277|       }
17278|   }
17279|   return NO_ERROR;
17280| }
17281|
17282| // EntryNum is the MFT * ClustersPerMFT
17283| // Datarun is BEGINNING of runs we have read so far
17284| // so if we need to find entry 16 (ClustersPerMFT=2)
17285|   | then
17286| // we need to find cluster offset 16*2 = 32.
17287| ULONGLONG FindClusterForEntryNum( PVOID DataRun,
17288|   | ULONGLONG EntryNum)
17289| {
17290|   ULARGE_INTEGER Cluster={0};
17291|   ULARGE_INTEGER Length={0};
17292|   ULONGLONG Run=0;
17293|   ULONG Offset=0;
17294|   ULONG Sparse=0;
17295|   ULONGLONG EntriesFound=0;
17296|   while(1) {
17297|       | if(GetRun(DataRun,&Offset,&Cluster,&Length,&Sparse)) {
17298|           //DLOG((TEXT("Offset = %12d,
17299|           | Cluster=%12l64d, Length=%12l64d,
17300|           | Sparse=%d\n"),Offset,Cluster, Length,Sparse));
17301|           if((EntryNum>=EntriesFound) &&
17302|           | (EntryNum<EntriesFound+Length.QuadPart)){
17303|               return
17304|               | (Cluster.QuadPart+EntryNum-EntriesFound) ;
17305|           }
17306|           EntriesFound += Length.QuadPart;

```

```

17297|     } else {
17298|         SetLastError(ERROR_INVALID_RUN);
17299|         return (ULONGLONG)-1;
17300|     }
17301| }
17302| }
17303|
17304| /* When joining extension data runs into a data run
   | complex we need to
17305| adjust the start LCN in the succeeding runs from
   | absolute to be
17306| relative to the starting LCN of the last run element
   | of the preceding data run.
17307| */
17308| ULONGLONG AdjustGapToRelative( PVOID DataRunComplex,
   | PVOID DataRun, PVOID DataRunComplexLimit)
17309| {
17310|     ULONG Offset=0;
17311|     ULONG Sparse=0;
17312|     CHAR Type = GetByte(DataRun,(Offset));
17313|     CHAR HoldChar;
17314|     SHORT HoldShort;
17315|     // TRIBYTE HoldTriByte;
17316|     LONG HoldLong;
17317|
17318|     // ULONGLONG LastLCNSoFar = (FindClusterForEntryNum(
   | DataRunComplex, ContinueVCN.QuadPart-1));
17319|     ULARGE_INTEGER LastLCNSoFar =
   | GetPackedDataRunFinalStartLCN( DataRunComplex,
   | DataRunComplexLimit );
17320|     if (LastLCNSoFar.QuadPart) {
17321|         (Sparse) = 0;
17322|
17323|         if(!Type)
17324|             return 0;
17325|
17326|         (Offset) += GetLengthLength(Type)+1;
17327|
17328|         switch(GetClusterLength(Type)) {
17329|             case 0: (Sparse) = 1;
17330|                 //Am assuming a datarun can't start
   | with a sparse - but I'm worried.
17331|                 //Obviously (maybe?) the first in
   | the complex can't but am not so
17332|                 //sure of the start of subsequent
   | extension dataruns!!
17333|                 //So I shouldnae be able to get
   | here and want to know if I do.
17334|                 _ASSERTE (FALSE);
17335|                 break;

```

```

17336| #if 0
17337|         case 1: PutChar(DataRun,(Offset),
| GetChar(DataRun,(Offset))-LastLCNSoFar.QuadPart);
17338|         break;
17339|         case 2: PutShort(DataRun,(Offset),
| GetShort(DataRun,(Offset))-LastLCNSoFar);
17340|         break;
17341|         case 3: PutTriByte(DataRun,(Offset),
| GetTriByte(DataRun,(Offset))-LastLCNSoFar);
17342|         break;
17343|         case 4: PutLong(DataRun,(Offset),
| GetLong(DataRun,(Offset))-LastLCNSoFar);
17344|         break;
17345| #else
17346|         case 1: HoldChar =
| (UCHAR)(GetChar(DataRun,(Offset))-LastLCNSoFar.QuadPart)
| ;
17347|         PutChar(DataRun,(Offset),HoldChar);
17348|         break;
17349|         case 2: HoldShort =
| (USHORT)(GetShort(DataRun,(Offset))-LastLCNSoFar.QuadPar
| t);
17350|         PutShort(DataRun,(Offset),HoldShort);
17351|         break;
17352|         case 3: HoldLong =
| (ULONG)(GetTriByte(DataRun,(Offset))-LastLCNSoFar.QuadPa
| rt);
17353|         PutTriByte(DataRun,(Offset),HoldLong);
17354|         break;
17355|         case 4: HoldLong =
| (ULONG)(GetLong(DataRun,(Offset))-LastLCNSoFar.QuadPart)
| ;
17356|         PutLong(DataRun,(Offset),HoldLong);
17357|         break;
17358| #endif
17359|         default:
17360|         DLOG((TEXT("Unable to decompress run
| length of %d\n"),GetClusterLength(Type)));
17361|         break;
17362|     }
17363|     return 1;
17364| } else {
17365|     return 0;
17366| }
17367| }
17368|
17369| /*
17370| returns how many entries will be required to hold

```



```

    | the VCN to LCN conversion table
17371| */
17372| ULONG GetDataRunLength( PVOID DataRun, ULONGLONG
    | NumClusters )
17373| {
17374|     ULARGE_INTEGER Cluster={0};
17375|     ULARGE_INTEGER Length={0};
17376|     ULONGLONG CurrentVCNEntry=0;
17377|     ULONG Offset=0;
17378|     ULONG Sparse=0;
17379|     ULONG Count=0;
17380|
17381|     while(CurrentVCNEntry<NumClusters) {
17382|
    | if(GetRun(DataRun,&Offset,&Cluster,&Length,&Sparse)) {
17383|         Count++;
17384|         CurrentVCNEntry+=Length.QuadPart;
17385|     } else {
17386|         SetLastError(ERROR_INVALID_RUN);
17387|         return 0;
17388|     }
17389| }
17390| return Count;
17391| }
17392|
17393| /*
17394| returns the length of the compressed data run
17395| */
17396| ULONG GetPackedDataRunLength( PVOID DataRun, ULONGLONG
    | NumClusters )
17397| {
17398|     ULARGE_INTEGER Cluster={0};
17399|     ULARGE_INTEGER Length={0};
17400|     ULONGLONG CurrentVCNEntry=0;
17401|     ULONG Offset=0;
17402|     ULONG Sparse=0;
17403|     ULONG Count=0;
17404|
17405|     while(CurrentVCNEntry<NumClusters) {
17406|
    | if(GetRun(DataRun,&Offset,&Cluster,&Length,&Sparse)) {
17407|         Count++;
17408|         CurrentVCNEntry+=Length.QuadPart;
17409| //         DLOG((TEXT(" %20d %20d
    | %20d \n"),Count,CurrentVCNEntry,Length.QuadPart));
17410|     } else {
17411|         SetLastError(ERROR_INVALID_RUN);
17412|         return 0;
17413|     }
17414| }

```

```

17415|   return Offset;
17416| }
17417|
17418| /*
17419|   returns the final StartLCN of the compressed data
17420|   | run
17421| */
17421| ULARGE_INTEGER GetPackedDataRunFinalStartLCN( PVOID
17422|   | DataRun, PVOID DataRunLimit)
17422| {
17423|   ULARGE_INTEGER Cluster={0};
17424|   ULARGE_INTEGER Length={0};
17425|   ULONGLONG CurrentVCNEntry=0;
17426|   ULONG Offset=0;
17427|   ULONG Sparse=0;
17428|   ULONG Count=0;
17429|
17430|   while((BYTE*)DataRun+Offset<(BYTE*)DataRunLimit) {
17431|
17432|     | if(GetRun(DataRun,&Offset,&Cluster,&Length,&Sparse)) {
17433|       Count++;
17434|       CurrentVCNEntry+=Length.QuadPart;
17435|       // DLOG((TEXT(" %20d %20d
17436|         | %20d \n"),Count,CurrentVCNEntry,Length.QuadPart));
17437|     } else {
17438|       SetLastError(ERROR_INVALID_RUN);
17439|       Cluster.QuadPart=0;
17440|       return Cluster;
17441|     }
17442|   }
17443|   return Cluster;
17444| }
17445|
17446| /*
17447|   Counts all the set bits(1) in a data block
17448|   */
17449| ULONGLONG CountSetBits( PVOID Data, ULONGLONG ByteCount
17450|   | )
17451| {
17452|   BYTE *Bytes = Data;
17453|   ULONGLONG i=0;
17454|   ULONGLONG Count=0;
17455|
17456|   while(i<ByteCount) {
17457|     Count += (Bytes[i] & 0x80 ? 1 : 0);
17458|     Count += (Bytes[i] & 0x40 ? 1 : 0);
17459|     Count += (Bytes[i] & 0x20 ? 1 : 0);
17460|     Count += (Bytes[i] & 0x10 ? 1 : 0);
17461|     Count += (Bytes[i] & 0x08 ? 1 : 0);
17462|     Count += (Bytes[i] & 0x04 ? 1 : 0);

```

```

17460|     Count += (Bytes[i] & 0x02 ? 1 : 0);
17461|     Count += (Bytes[i] & 0x01 ? 1 : 0);
17462|     i++;
17463| }
17464| return Count;
17465| }
17466|
17467| /*
17468|  Counts all the clear(0) bits in a data block
17469| */
17470| ULONGLONG CountClearBits( PVOID Data, ULONGLONG
    | ByteCount )
17471| {
17472|     BYTE *Bytes = Data;
17473|     ULONGLONG i=0;
17474|     ULONGLONG Count=0;
17475|
17476|     while(i<ByteCount) {
17477|         Count += (Bytes[i] & 0x80 ? 0 : 1);
17478|         Count += (Bytes[i] & 0x40 ? 0 : 1);
17479|         Count += (Bytes[i] & 0x20 ? 0 : 1);
17480|         Count += (Bytes[i] & 0x10 ? 0 : 1);
17481|         Count += (Bytes[i] & 0x08 ? 0 : 1);
17482|         Count += (Bytes[i] & 0x04 ? 0 : 1);
17483|         Count += (Bytes[i] & 0x02 ? 0 : 1);
17484|         Count += (Bytes[i] & 0x01 ? 0 : 1);
17485|         i++;
17486|     }
17487|     return Count;
17488| }
17489|
17490| /*
17491|  Calculates how much free space is available. It
    | does this by opening the bitmap file
17492|  (FILE_BITMAP) and reading the DATA_ATTR data, which
    | is a bitmap of which clusters are
17493|  in use. 1 means in use, 0 means free.
17494| */
17495| ULONGLONG NTFS_GetNumFreeClusters( pVolumeInfo
    | VolumeInfo )
17496| {
17497|     PNTFS_File File;
17498|     ULONG Err = 0;
17499|     ULONGLONG NumClusters = 0;
17500|     ULONG ReadSize = 32768;
17501|
17502|     File = NTFS_OpenFileByNumber( VolumeInfo,
    | FILE_BITMAP, DATA_ATTR, NULL );
17503|     if(File) {
17504|         ULONGLONG FileSize =

```

```

    | NTFS_GetFileSize(File);
17505|     BYTE      *Data      =
    | SAFE_Alloc(ReadSize);
17506|     ULONGLONG BytePos    = 0;
17507|
17508|     if(Data) {
17509|         while(BytePos<FileSize) {
17510|             if(BytePos+ReadSize>FileSize)
17511|                 ReadSize =
    | (ULONG)(FileSize-BytePos);
17512|
17513|             Err = NTFS_ReadFile(File, BytePos,
    | ReadSize, Data);
17514|             if(!Err) {
17515|                 NumClusters += CountClearBits(Data,
    | ReadSize);
17516|                 BytePos+=ReadSize;
17517|             } else {
17518|                 NumClusters = (ULONGLONG)-1;
17519|                 DLOG((TEXT("Error %08x reading
    | attribute data\n"),Err));
17520|                 SetLastError(Err);
17521|                 break;
17522|             }
17523|         }
17524|         SAFE_Free(Data);
17525|     } else {
17526|         DLOG((TEXT("Error, out of memory\n")));
17527|         SetLastError(ERROR_OUTOFMEMORY);
17528|     }
17529|     NTFS_CloseFile(File);
17530| } else {
17531|     DLOG((TEXT("Unable to open file,
    | error=%08x\n"),GetLastError()));
17532| }
17533| return NumClusters;
17534| }
17535|
17536| /*
17537| Reads the cluster specified
17538| */
17539| ULONG NTFS_ReadLogicalCluster( pVolumeInfo VolumeInfo,
    | ULONGLONG Cluster, ULONG NumClusters, PVOID Buffer )
17540| {
17541|     ULONG Err;
17542|     //DLOG((TEXT("Reading sector %12I64d
    | (Cluster=%12I64d)\n"),NTFS_ClusterToPhysical(VolumeInfo,
    | Cluster),Cluster));
17543|     Err = DASD_Read( VolumeInfo->LockHandle,
17544|

```

```

    | NTFS_ClusterToPhysical(VolumeInfo,Cluster),
17545|
    | VolumeInfo->NtfsBootSector->SectorsPerCluster*NumCluster
    | s,
17546|         Buffer
17547|     );
17548|     return Err;
17549| }
17550|
17551| /*
17552|     Writes the cluster specified
17553| */
17554| ULONG NTFS_WriteLogicalCluster( pVolumeInfo VolumeInfo,
    | ULONGLONG Cluster, ULONG NumClusters, PVOID Buffer )
17555| {
17556|     ULONG Err;
17557|     //DLOG((TEXT("Writing sector %12I64d
    | (Cluster=%12I64d)\n"),NTFS_ClusterToPhysical(VolumeInfo,
    | Cluster),Cluster));
17558|     Err = DASD_Write( VolumeInfo->LockHandle,
17559|
    | NTFS_ClusterToPhysical(VolumeInfo,Cluster),
17560|
    | VolumeInfo->NtfsBootSector->SectorsPerCluster*NumCluster
    | s,
17561|         Buffer
17562|     );
17563|     return Err;
17564| }
17565|
17566| /*
17567|     Gets the attribute by its type number. Usually
    | only called with well known ids such as
17568|     ATTR_DATA, ATTR_FILENAME, etc...
17569| */
17570| PVOID NTFS_GetAttributeByType( pVolumeInfo VolumeInfo,
    | PMFT Mft, ULONG Type )
17571| {
17572|     if(Mft->Signature == 'ELIF') {
17573|
17574|         PATTRIBUTE Attribute =
    | (PATTRIBUTE)&(((char*)Mft)[Mft->OffsetToAttributes]);
17575|         ULONG Where = 0;
17576|
17577|         while(Attribute->Type!=0xffffffff) {
17578|             if(Attribute->Type==Type) {
17579|                 return Attribute;
17580|             }
17581|             Where += Attribute->Length;
17582|

```

```

17583|         Attribute =
17584|         | (PATATTRIBUTE)&(((char*)Mft)[Mft->OffsetToAttributes+Where
17585|         | ]);
17586|     }
17587| } else {
17588|     DLOG((TEXT("Not an Mft!\n")));
17589| }
17590| return NULL;
17591| }
17592| /*
17593|  Gets the attribute by its name
17594| */
17595| PVOID NTFS_GetAttributeByName( pVolumeInfo VolumeInfo,
17596|     | PMFT Mft, ULONG Type, WCHAR *Name )
17597| {
17598|     if(Name) {
17599|         if(Mft->Signature == 'ELIF') {
17600|             PATATTRIBUTE Attribute =
17601|             | (PATATTRIBUTE)&(((char*)Mft)[Mft->OffsetToAttributes]);
17602|             ULONG Where = 0;
17603|             while(Attribute->Type!=0xffffffff) {
17604|                 WCHAR *CurrentAttrName =
17605|                 | NTFS_GetAttributeNamePointer(Attribute);
17606|                 if(CurrentAttrName) {
17607|                     if((Attribute->Type==Type) &&
17608|                     | (wcsncmp(CurrentAttrName,Name,Attribute->NameLength)==0)
17609|                     | )
17610|                         return Attribute;
17611|                     }
17612|                     Where += Attribute->Length;
17613|                     Attribute =
17614|                     | (PATATTRIBUTE)&(((char*)Mft)[Mft->OffsetToAttributes+Where
17615|                     | ]);
17616|                 }
17617|             } else {
17618|                 DLOG((TEXT("Not an Mft!\n")));
17619|             }
17620|         } else {
17621|             return NTFS_GetAttributeByType( VolumeInfo,
17622|             | Mft, Type );
17623|         }
17624|     }
17625|     return NULL;
17626| }
17627| /*
17628|  Read the Mft entry of the specified file

```

```

17623| */
17624| ULONG NTFS_ReadMftEntryNumberRaw( pVolumeInfo
    | VolumeInfo, ULONGLONG EntryNum, PVOID Buffer )
17625| {
17626|     _ASSERT(EntryNum!=0);
17627|     return NTFS_ReadFile( VolumeInfo->MftFile,
17628|         | (EntryNum*VolumeInfo->NtfsBootSector->MFTRecordSize) *
        | NTFS_ClusterSizeInBytes(VolumeInfo),
17629|         | NTFS_MftByteSize(VolumeInfo),
17630|         Buffer );
17631| }
17632|
17633| /*
17634|     Read the Mft entry of the specified file
17635| */
17636| ULONG NTFS_ReadMftEntryNumber( pVolumeInfo VolumeInfo,
    | ULONGLONG EntryNum, PVOID Buffer )
17637| {
17638|     if(EntryNum==0) {
17639|         // Mft of mft is always in memory..
17640|         // this has to be the case, because
        | NTFS_ReadFile calls us, and this prevents
17641|         // a circle
17642|         | memcpy(Buffer,VolumeInfo->Mft,NTFS_MftByteSize(VolumeInf
        | o));
17643|         return 0;
17644|     } else {
17645|
17646|         ULONG Err =
        | NTFS_ReadMftEntryNumberRaw(VolumeInfo,EntryNum,Buffer);
17647|
17648|         if(!Err) {
17649|             // fixup mft entry
17650|             if(*(ULONG*)Buffer==(ULONG)'ELIF') {
17651|                 if(!NTFS_FixupRecord( VolumeInfo,
        | Buffer )) {
17652|                     DLOG((TEXT("Error fixing up Mft
        | entry!!!\n")));
17653|                 }
17654|             } else {
17655|                 DLOG((TEXT("Didnt read a mft!\n")));
17656|             }
17657|         }
17658|         return Err;
17659|     }
17660| }
17661|

```

```

17662|
17663| /*
17664|   Writes the Mft entry of the specified file
17665| */
17666| ULONG NTFS_WriteMftEntryNumber( pVolumeInfo VolumeInfo,
    | ULONGLONG EntryNum, PVOID Buffer )
17667| {
17668|   if(EntryNum==0) {
17669|     // Mft of mft is always in memory..
17670|     // this has to be the case, because
    | NTFS_ReadFile calls us, and this prevents
17671|     // a circle
17672|     DLOG((TEXT("Saving mft 0 is not
    | supported!!\n")));
17673|     return ERROR_NOT_SUPPORTED;
17674|   } else {
17675|     ULONG Err=ERROR_INVALID_PARAMETER;
17676|
17677|     if(*(ULONG*)Buffer==(ULONG)'ELIF') {
17678|       SAFE_ReadWrite(Buffer);
17679|       // undo the fix up so we can commit to disk
17680|       if(NTFS_UnfixupRecord( VolumeInfo, Buffer
    | )) {
17681|         Err = NTFS_WriteFile(
    | VolumeInfo->MftFile,
17682|         | (EntryNum*VolumeInfo->NtfsBootSector->MFTRecordSize) *
    | NTFS_ClusterSizeInBytes(VolumeInfo),
17683|         | NTFS_MftByteSize(VolumeInfo),
17684|         | Buffer );
17685|         // Fix the record up so it is readable
17686|         | if(!NTFS_FixupRecord(VolumeInfo,Buffer)) {
17687|           DLOG((TEXT("data buffer failed
    | check\n")));
17688|           Err = ERROR_DISK_CORRUPT;
17689|         }
17690|       } else {
17691|         DLOG((TEXT("Error fixing up Mft
    | entry!!!\n")));
17692|       }
17693|       SAFE_ReadOnly(Buffer);
17694|     } else {
17695|       DLOG((TEXT("Didnt read a mft!\n")));
17696|     }
17697|
17698|     return Err;
17699|   }
17700| }

```



```

17701|
17702|
17703| // TODO: Get the numbers associated with the attributes
      | instead of being hard coded
17704| void NTFS_GetAttributeNames( pVolumeInfo VolumeInfo )
17705| {
17706| }
17707|
17708| /*
17709|   This inits a volume so we can access it. This must
      | be called before any other
17710|   routines are called. NTFS_CloseVolume should be
      | called when done.
17711|   Sector should be the start of the volume
17712| */
17713| pVolumeInfo NTFS_OpenVolume( tDASDHandle *LockHandle,
      | ULONGLONG Sector )
17714| {
17715|   ULONG Err;
17716|   pVolumeInfo VolumeInfo=NULL;
17717|
17718|   VolumeInfo =
      | (pVolumeInfo)SAFE_Alloc(sizeof(tVolumeInfo));
17719|   if(VolumeInfo) {
17720|     memset(VolumeInfo,0,sizeof(tVolumeInfo));
17721|
17722|     VolumeInfo->VolumeStartSector = Sector;
17723|     VolumeInfo->LockHandle = LockHandle;
17724|     VolumeInfo->NtfsBootSector =
      | (PNTFS_BOOT_SECTOR)SAFE_Alloc(sizeof(NTFS_BOOT_SECTOR));
17725|     if(VolumeInfo->NtfsBootSector) {
17726|       // cant call NTFS_ReadFile yet as the mft
      | hasn't been read
17727|       //DLOG((TEXT("Reading sector %12I64d (NTFS
      | Boot Sector)\n"),Sector));
17728|       Err = DASD_Read( LockHandle, Sector, 1,
      | VolumeInfo->NtfsBootSector );
17729|       if(!Err) {
17730|         // Fixup the MftRecordSize for 4k
      | clusters
17731|
      | if(VolumeInfo->NtfsBootSector->MFTRecordSize==0xf6) {
17732|           // f6 is assumed to be .25 or 1k,
      | this info comes from the linux ntfs
17733|           // driver, i havent seen it. Assume
      | Cluster Size.
17734|           DLOG((TEXT("Modifying MFTRecordSize
      | from
      | %02x\n"),VolumeInfo->NtfsBootSector->MFTRecordSize));
17735|

```

```

    | DumpNtfsBootSector(VolumeInfo->NtfsBootSector);
17736|
    | VolumeInfo->NtfsBootSector->MFTRecordSize = max(1024 /
    | VolumeInfo->NtfsBootSector->BytesPerSector,VolumeInfo->N
    | tfsBootSector->SectorsPerCluster);
17737|         } else
17738|
    | if(VolumeInfo->NtfsBootSector->MFTRecordSize>0x80) {
17739|         // assume something wrong
17740|         DLOG((TEXT("Modifying MFTRecordSize
    | from %02x - Something is
    | wrong!\n"),VolumeInfo->NtfsBootSector->MFTRecordSize));
17741|
    | DumpNtfsBootSector(VolumeInfo->NtfsBootSector);
17742|
    | VolumeInfo->NtfsBootSector->MFTRecordSize = max(1024 /
    | VolumeInfo->NtfsBootSector->BytesPerSector,VolumeInfo->N
    | tfsBootSector->SectorsPerCluster);
17743|         }
17744|
17745|         VolumeInfo->Mft =
    | (PMFT)SAFE_Alloc(NTFS_MftByteSize(VolumeInfo));
17746|         if(VolumeInfo->Mft) {
17747|             // cant call NTFS_ReadFile yet as
    | the mft hasn't been read
17748|             // NTFS_ReadLogicalCluster does not
    | require the Mft
17749|             Err = NTFS_ReadLogicalCluster(
    | VolumeInfo,
17750|
    | VolumeInfo->NtfsBootSector->MftCluster.QuadPart,
17751|
    | VolumeInfo->NtfsBootSector->MFTRecordSize,
17752|
    | VolumeInfo->Mft );
17753|             if(!Err) {
17754|                 if(VolumeInfo->Mft->Signature
    | == 'ELIF') {
17755|                     if(NTFS_FixupRecord(
    | VolumeInfo, VolumeInfo->Mft )) {
17756|                         VolumeInfo->MftFile =
    | NTFS_OpenFileByNumber( VolumeInfo, FILE_MFT, DATA_ATTR,
    | NULL );
17757|                         if(VolumeInfo->MftFile)
    | {
17758|                             // get the unicode
    | uppercase table, needed for directory
17759|                             // searches
17760|                             PNTFS_File
    | File=NTFS_OpenFileByNumber(VolumeInfo, FILE_UPCASE,

```



```

17788|                } else {
17789|
17790|                | DLOG((TEXT("Unable to open upcase file\n")));
17791|                }
17792|                | NTFS_CloseFile(VolumeInfo->MftFile);
17793|                } else {
17794|                DLOG((TEXT("Unable
17795|                | to open mft\n")));
17796|                }
17797|                } else {
17798|                DLOG((TEXT("Mft fixup
17799|                | failed\n")));
17800|                }
17801|                } else {
17802|                DLOG((TEXT("Sector %08x is
17803|                | not the
17804|                | Mft!\n"),NTFS_ClusterToPhysical(VolumeInfo,VolumeInfo->N
17805|                | tfsBootSector->MftCluster.QuadPart)));
17806|                }
17807|                } else {
17808|                DLOG((TEXT("Error %08x reading
17809|                | cluster
17810|                | %08x\n"),Err,VolumeInfo->NtfsBootSector->MftCluster.Quad
17811|                | Part));
17812|                }
17813|                SAFE_Free(VolumeInfo->Mft);
17814|                } else {
17815|                DLOG((TEXT("Out of memory
17816|                | (Need=%d)\n"),NTFS_MftByteSize(VolumeInfo)));
17817|                }
17818|                } else {
17819|                DLOG((TEXT("Error %08x reading sector
17820|                | %08x\n"),Err,Sector));
17821|                }
17822|                SAFE_Free(VolumeInfo->NtfsBootSector);
17823|                } else {
17824|                DLOG((TEXT("Out of memory
17825|                | (Need=%d)\n"),sizeof(NTFS_BOOT_SECTOR)));
17826|                }
17827|                SAFE_Free(VolumeInfo);
17828|                } else {
17829|                DLOG((TEXT("Out of memory
17830|                | (Need=%d)\n"),sizeof(tVolumeInfo)));
17831|                }
17832|                return NULL;
17833|        }
17834|        /*

```

```

17824|   Frees memory associated with NTFS_OpenVolume
17825| */
17826| void NTFS_CloseVolume( pVolumeInfo VolumeInfo )
17827| {
17828|     if(VolumeInfo) {
17829|         if(VolumeInfo->UppcaseTable)
17830|             SAFE_Free(VolumeInfo->UppcaseTable);
17831|         if(VolumeInfo->MftFile)
17832|             NTFS_CloseFile(VolumeInfo->MftFile);
17833|         if(VolumeInfo->Mft)
17834|             SAFE_Free(VolumeInfo->Mft);
17835|         if(VolumeInfo->NtfsBootSector)
17836|             SAFE_Free(VolumeInfo->NtfsBootSector);
17837|         SAFE_Free(VolumeInfo);
17838|     }
17839| }
17840|
17841|
17842| static void ReadPartitionTable ( tDASDHandle
    | *LockHandle, ULONGLONG Sector )
17843| {
17844|     ULONG i;
17845|     tMBR MBR;
17846|     ULONG Err;
17847|     pVolumeInfo VolumeInfo;
17848|
17849|     DLOG((TEXT("Reading sector %12l64d (Partition
    | Table)\n"),Sector));
17850|     Err = DASD_Read ( LockHandle, Sector, 1, &MBR );
17851|     for(i=0;i<4;i++) {
17852|         if(MBR.Part[i].SystemType == PARTITION_NTFS ) {
17853|             VolumeInfo = NTFS_OpenVolume( LockHandle,
    | MBR.Part[i].StartSector+Sector );
17854|             if(VolumeInfo) {
17855|                 DumpNTFSPartition( VolumeInfo );
17856|
17857|                 NTFS_CloseVolume( VolumeInfo );
17858|             }
17859|         } else
17860|             if(MBR.Part[i].SystemType ==
    | PARTITION_DOSExtended ) {
17861|                 // recursively call ourselves
17862|                 ReadPartitionTable( LockHandle,
    | MBR.Part[i].StartSector );
17863|             }
17864|     }
17865| }
17866|
17867| static void ReadDrive( ULONG DriveNum )
17868| {

```

```

17869|  tDASDHandle *LockHandle;
17870|
17871|  LockHandle = DASD_LockDevice( DriveNum );
17872|  if(LockHandle) {
17873|      ReadPartitionTable( LockHandle, 0 );
17874|
17875|      DASD_UnlockDevice( LockHandle );
17876|  } else {
17877|      DLOG((TEXT("Unable to obtain lock to
    | device\n")));
17878|  }
17879| }
17880|
17881|
17882| PNTFS_FIDO_INFO NTFS_FidoFirstFile( pVolumeInfo
    | VolumeInfo, WCHAR* lpFileName, LPWIN32_FIND_DATA
    | lpFindFileData )
17883| {
17884|     tNTFS_FIDO_INFO *FI;
17885|
17886|     FI = SAFE_Alloc(sizeof(tNTFS_FIDO_INFO));
17887|     if(FI) {
17888|         FI->DirectoryFile = NTFS_OpenFileByName(
    | VolumeInfo, lpFileName, FILENAME_ATTR, NULL );
17889|         if(FI->DirectoryFile) {
17890|             // Before returning, return back the first
    | file and store the index of it.
17891|             if (NTFS_FidoNextFile( FI, lpFindFileData
    | )) {
17892|                 return FI;
17893|             } else {
17894|                 // error set from Find Next
17895|             }
17896|         } else {
17897|             // error set from call to open
17898|         }
17899|         SAFE_Free(FI);
17900|     } else {
17901|         SetLastError(ERROR_OUTOFMEMORY);
17902|     }
17903|     return NULL;
17904| }
17905|
17906| BOOL NTFS_FidoNextFile( PNTFS_FIDO_INFO FI,
    | LPWIN32_FIND_DATA lpFindFileData )
17907| {
17908|     if (FindEntryNumInDir(
    | FI->DirectoryFile->VolumeInfo,
    | FI->DirectoryFile->MftEntryNum, FI->IndexesFound,
    | lpFindFileData )){

```

```

17909|     FI->IndexesFound++;
17910|     return TRUE;
17911| } else {
17912|     //already set or eof???
17913|     | SetLastError(ERROR_NOT_SUPPORTED);
17914|     return FALSE;
17915| }
17916|
17917| BOOL NTFS_FidoClose( PNTFS_FIDO_INFO hFindFile )
17918| {
17919|     BOOL B=FALSE;
17920|     if(hFindFile) {
17921|         if(hFindFile->DirectoryFile) {
17922|             NTFS_CloseFile(hFindFile->DirectoryFile);
17923|             B = TRUE;
17924|         } else {
17925|             SetLastError(ERROR_FILE_INVALID);
17926|         }
17927|         SAFE_Free(hFindFile);
17928|     } else {
17929|         SetLastError(ERROR_FILE_INVALID);
17930|     }
17931|     return B;
17932| }
17933|
17934|
17935| PNTFS_FIND_INFO NTFS_FindFirstFile( pVolumeInfo
17936|     | VolumeInfo, WCHAR* lpFileName, LPWIN32_FIND_DATA
17937|     | lpFindFileData )
17938| {
17939|     tNTFS_FIND_INFO *FI;
17940|     FI = SAFE_Alloc(sizeof(tNTFS_FIND_INFO));
17941|     if(FI) {
17942|         FI->DirectoryFile = NTFS_OpenFileByName(
17943|             | VolumeInfo, lpFileName, FILENAME_ATTR, NULL );
17944|         if(FI->DirectoryFile) {
17945|             if (OpenIndexRoot( FI, VolumeInfo,
17946|                 | FI->DirectoryFile->MftEntryNum, lpFindFileData )) {
17947|                 // Before returning, find the first
17948|                 | file.
17949|                 if (NTFS_FindNextFile( FI,
17950|                     | lpFindFileData )) {
17951|                     return FI;
17952|                 } else {
17953|                     DLOG((TEXT("Unable to find first file
17954|                         | after opening index root\n")));
17955|                 }
17956|             } else {

```

```

17951|         DLOG((TEXT("Unable to open index
    | root\n")));
17952|     }
17953| } else {
17954|     DLOG((TEXT("Unable to open requested
    | directory\n")));
17955| }
17956|     SAFE_Free(FI);
17957| } else {
17958|     SetLastError(ERROR_OUTOFMEMORY);
17959|     DLOG((TEXT("Out of memory for NTFS_FIND_INFO
    | structure\n")));
17960| }
17961|     return NULL;
17962| }
17963|
17964| BOOL NTFS_FindNextFile( PNTFS_FIND_INFO FI,
    | LPWIN32_FIND_DATA lpFindFileData )
17965| {
17966|     BOOL SteppedStatus;
17967|     while ((SteppedStatus = StepToNextIndex( FI,
    | FI->DirectoryFile->VolumeInfo, lpFindFileData )) &&
    | (FI->DeepestNode->IE->FileNameType == FILENAME_DOS)) {
17968|         // Bypass DOS names - we'll get the files
    | anyway by their NT name
17969|         | ((BYTE*)FI->DeepestNode->IE)+=NTFS_IndexEntrySize(FI->De
    | epestNode->IE);
17970|     }
17971|     if (SteppedStatus) {
17972|         // Give him the lowdown ...
17973|         lpFindFileData->dwFileAttributes = (DWORD)
    | FI->DeepestNode->IE->Attributes.QuadPart;
17974|     #if 0
17975|         lpFindFileData->ftCreationTime = (unsigned
    | __int64)FI->DeepestNode->IE->FileCreationTime;
17976|         lpFindFileData->ftLastAccessTime =
    | (FILETIME)FI->DeepestNode->IE->LastAccessTime;
17977|         lpFindFileData->ftLastWriteTime =
    | (FILETIME)FI->DeepestNode->IE->LastModifiedTime;
17978|     #else
17979|         lpFindFileData->ftCreationTime =
    | FI->DeepestNode->IE->FileCreationTime;
17980|         lpFindFileData->ftLastAccessTime =
    | FI->DeepestNode->IE->LastAccessTime;
17981|         lpFindFileData->ftLastWriteTime =
    | FI->DeepestNode->IE->LastModifiedTime;
17982|     #endif
17983|         lpFindFileData->nFileSizeHigh =
    | FI->DeepestNode->IE->AttributeSize.HighPart;

```



```

17984|    lpFindFileData->nFileSizeLow    =
    | FI->DeepestNode->IE->AttributeSize.LowPart;
17985|    wcsncpy
    | (lpFindFileData->cFileName,FI->DeepestNode->IE->FileName
    | ,FI->DeepestNode->IE->FileNameLength );
17986|
    | lpFindFileData->cFileName[FI->DeepestNode->IE->FileNameL
    | ength] = 0;
17987| //    lpFindFileData->cAlternateFileName = NULL;
17988|    lpFindFileData->cAlternateFileName[0] = 0;
17989| // TCHAR    cFileName[ MAX_PATH ]; Why this comment -
    | to remind me to sort out WCHAR vs. TCHAR ??
17990|    // ... and move on.
17991|
    | ((BYTE*)FI->DeepestNode->IE)+=NTFS_IndexEntrySize(FI->De
    | epestNode->IE);
17992| }
17993| return (SteppedStatus);
17994| }
17995|
17996| BOOL NTFS_FindClose( PNTFS_FIND_INFO hFindFile )
17997| {
17998|     BOOL B=FALSE;
17999|     if(hFindFile) {
18000|         while (((BYTE*)hFindFile->DeepestNode !=
    | (BYTE*)&hFindFile->DeepestNode)) {
18001|             PNODE NodeElect = hFindFile->DeepestNode;
18002|             SAFE_Free(hFindFile->DeepestNode->IA);
18003|             SAFE_Free(hFindFile->DeepestNode);
18004|             hFindFile->DeepestNode = NodeElect;
18005|         }
18006|         SAFE_Free(hFindFile->IR);
18007|         if(hFindFile->IAFile) {
18008|             NTFS_CloseFile(hFindFile->IAFile);
18009|         }
18010|         if(hFindFile->IRFile) {
18011|             NTFS_CloseFile(hFindFile->IRFile);
18012|             if(hFindFile->DirectoryFile) {
18013|
    | NTFS_CloseFile(hFindFile->DirectoryFile);
18014|                 B = TRUE;
18015|             } else {
18016|                 DLOG((TEXT("No directory to close
    | find\n")));
18017|                 SetLastError(ERROR_FILE_INVALID);
18018|             }
18019|         } else {
18020|             SetLastError(ERROR_FILE_INVALID);
18021|         }
18022|         SAFE_Free(hFindFile);

```

```

18023|    } else {
18024|        DLOG((TEXT("No NTFS_FIND_INFO to close
| find\n")));
18025|        SetLastError(ERROR_FILE_INVALID);
18026|    }
18027|    return B;
18028| }
18029|
18030| BOOL NTFS_GetFileSecurity( WCHAR* lpFileName, //
| address of string for file name
18031|        SECURITY_INFORMATION
| RequestedInformation, // requested information
18032|        PSECURITY_DESCRIPTOR
| pSecurityDescriptor, // address of security descriptor
18033|        DWORD nLength, // size of
| security descriptor buffer
18034|        LPDWORD lpnLengthNeeded //
| address of required size of buffer
18035|    )
18036| {
18037|     SetLastError(ERROR_NOT_SUPPORTED);
18038|     return FALSE;
18039| }
18040|
18041| DWORD NTFS_GetFullPathName( WCHAR* lpFileName, //
| address of name of file to find path for
18042|        DWORD nBufferLength, //
| size, in characters, of path buffer
18043|        WCHAR* lpBuffer, // address
| of path buffer
18044|        WCHAR** lpFilePart // address
| of filename in path
18045|    )
18046| {
18047|     SetLastError(ERROR_NOT_SUPPORTED);
18048|     return 0;
18049| }
18050|
18051|
18052| /*
18053| BackupRead
18054| BackupSeek
18055| BackupWrite
18056| CloseFile
18057| CopyFile
18058| CreateDirectory
18059| CreateDirectoryEx
18060| CreateFile
18061| DeleteFile
18062| FindClose

```

```

18063| FindFirstFile
18064| FindNextFile
18065| FlushFileBuffers
18066| GetCompressedFileSize
18067| GetCurrentDirectory
18068| GetFileAttributes
18069| GetFileInformationByHandle
18070| GetFileSecurity
18071| GetFileSize
18072| GetFileTime
18073| GetFullPathName
18074| MoveFile
18075| MoveFileEx
18076| ReadFile
18077| ReadFileEx
18078| RemoveDirectory
18079| RenameFile
18080| SetCurrentDirectory
18081| SetEndOfFile
18082| SetFileAttributes
18083| SetFilePointer
18084| SetFileTime
18085| WriteFile
18086| WriteFileEx
18087| */
18088|
18089| static void DisplayHelp( void )
18090| {
18091|     DLOG((TEXT("-drive:#\n")));
18092|     DLOG((TEXT("\twhere # is drive number starting with
| 0\n")));
18093|     DLOG((TEXT("-part:#\n")));
18094|     DLOG((TEXT("\twhere # is partition number from
| -listparts\n")));
18095|     DLOG((TEXT("-listparts\n")));
18096|     DLOG((TEXT("\tLists partitions on drive\n")));
18097|     DLOG((TEXT("-asmft:#\n")));
18098|     DLOG((TEXT("\tDisplay Cluster # as an mft\n")));
18099|     DLOG((TEXT("-asntfsboot:#\n")));
18100|     DLOG((TEXT("\tDisplays Sector # as a ntfs boot
| sector\n")));
18101|     DLOG((TEXT("-walk:#\n")));
18102|     DLOG((TEXT("\tDoes a directory of mft entry
| #\n")));
18103|     DLOG((TEXT("-dir:<dir>\n")));
18104|     DLOG((TEXT("\tDoes a directory of dir <dir>\n")));
18105|     DLOG((TEXT("-nokey\n")));
18106|     DLOG((TEXT("\tDoesnt pause for a key press at end
| of run\n")));
18107|     DLOG((TEXT("-memcheck\n")));

```

```

18108| DLOG((TEXT("\tDisplays memory usage at end of
| run\n")));
18109| DLOG((TEXT("-free\n")));
18110| DLOG((TEXT("\tDisplays clusters free and
| available\n")));
18111| DLOG((TEXT("-copy:#[stream]:<file>\n")));
18112| DLOG((TEXT("\tCopys attribute # and optionally
| named attribute [stream] for file <file> to file
| 'data\n")));
18113| DLOG((TEXT("-findfile:<File>\n")));
18114| DLOG((TEXT("\treturns the mft entry of file
| <file>\n")));
18115| DLOG((TEXT("-mftentry:#\n")));
18116| DLOG((TEXT("\tDumps mft entry #\n")));
18117| DLOG((TEXT("-createstream\n")));
18118| DLOG((TEXT("\tCreates a multistream file\n")));
18119|
18120| }
18121|
18122| typedef struct _FILE_END_OF_FILE_INFORMATION {
18123|     LARGE_INTEGER EndOfFile;
18124| } FILE_END_OF_FILE_INFORMATION,
| *PFILE_END_OF_FILE_INFORMATION;
18125|
18126|
18127| ULONG NtDeleteFile( WCHAR *FileName )
18128| {
18129|     HANDLE Handle;
18130|     ULONG Err;
18131|     UNICODE_STRING Uni;
18132|     OBJECT_ATTRIBUTES ObjectAttributes={0};
18133|     IO_STATUS_BLOCK IoStatus={0};
18134|
18135|     RtlInitUnicodeString( &Uni, FileName);
18136|
18137|     InitializeObjectAttributes ( &ObjectAttributes,
18138|                                &Uni,
18139|                                OBJ_CASE_INSENSITIVE,
18140|                                NULL,
18141|                                NULL );
18142|     Err = NtCreateFile( &Handle,
18143|                        DELETE | SYNCHRONIZE ,
18144|                        // desired access
18145|                        &ObjectAttributes,
18146|                        // object attributes
18147|                        &IoStatus,
18148|                        NULL,
18149|                        // alloc size
18150|                        FILE_ATTRIBUTE_NORMAL,
18151|                        // file attributes

```

```

18148|          0, // share access
18149|          FILE_OPEN,
    | // create disposition
18150|          FILE_SYNCHRONOUS_IO_NONALERT,
    | // create options
18151|          NULL,
    | // eabuffer
18152|          0 );
    | // ealength
18153|
18154|  if(!Err) {
18155|      FILE_DISPOSITION_INFORMATION FSInfo;
18156|      FSInfo.DeleteFile = 1;
18157|
18158|      Err = NtSetInformationFile(
18159|          Handle,          // IN HANDLE
    | FileHandle,
18160|          &IoStatus,      // OUT
    | PIO_STATUS_BLOCK IoStatusBlock,
18161|          &FSInfo,        // IN PVOID
    | FileInformation,
18162|          sizeof(FILE_DISPOSITION_INFORMATION),
    | // IN ULONG Length,
18163|          FileDispositionInformation// IN
    | FILE_INFORMATION_CLASS FileInformationClass
18164|      );
18165|
18166|      NtClose(Handle);
18167|  } else {
18168|      DLOG((TEXT("Error %08x deleting file
    | '%s\\n"),Err,FileName));
18169|  }
18170|  return Err;
18171| }
18172|
18173| ULONG AllocSpace( WCHAR *VolumeName, WCHAR *FileName,
    | LARGE_INTEGER AllocSize )
18174| {
18175|  HANDLE Handle;
18176|  ULONG Err;
18177|  WCHAR FullName[256];
18178|  UNICODE_STRING Uni;
18179|  OBJECT_ATTRIBUTES ObjectAttributes={0};
18180|  IO_STATUS_BLOCK IoStatus={0};
18181|
18182|  swprintf(FullName, L"%s%s",VolumeName,FileName);
18183|
18184|  NtDeleteFile(FullName);
18185|
18186|  RtlInitUnicodeString( &Uni, FullName);

```

```

18187|
18188|    InitializeObjectAttributes ( &ObjectAttributes,
18189|                                &Uni,
18190|                                OBJ_CASE_INSENSITIVE,
18191|                                NULL,
18192|                                NULL );
18193|    Err = NtCreateFile( &Handle,
18194|                        FILE_GENERIC_READ |
18195|                        | FILE_GENERIC_WRITE,          // desired access
18196|                        &ObjectAttributes,
18197|                        | // object attributes
18198|                        &IoStatus,
18199|                        NULL,
18200|                        | // alloc size
18201|                        FILE_ATTRIBUTE_NORMAL,
18202|                        | // file attributes
18203|                        0, // share access
18204|                        FILE_CREATE,
18205|                        | // create disposition
18206|                        FILE_SYNCHRONOUS_IO_NONALERT,
18207|                        | // create options
18208|                        NULL,
18209|                        | // eabuffer
18210|                        0 );
18211|                        | // ealength
18212|
18213|    if(!Err) {
18214|        FILE_END_OF_FILE_INFORMATION FSInfo;
18215|        FSInfo.EndOfFile = AllocSize;
18216|
18217|        Err = NtSetInformationFile(
18218|            Handle,          // IN HANDLE
18219|            | FileHandle,
18220|            &IoStatus,          // OUT
18221|            | PIO_STATUS_BLOCK IoStatusBlock,
18222|            &FSInfo,          // IN PVOID
18223|            | FileInformation,
18224|            sizeof(FILE_END_OF_FILE_INFORMATION),
18225|            | // IN ULONG Length,
18226|            FileEndOfFileInformation// IN
18227|            | FILE_INFORMATION_CLASS FileInformationClass
18228|        );
18229|
18230|        NtClose(Handle);
18231|    } else {
18232|        DLOG((TEXT("Error %08x allocating %!64d bytes for
18233|            | file for '%s'\n"),Err,AllocSize,FullName));
18234|    }
18235|    return Err;
18236| }

```

```

18223|
18224| ULONG InitializeSpace( tVolumeInfo *VolumeInfo, WCHAR
    | *FileName, LARGE_INTEGER AllocSize )
18225| {
18226|     ULONG Err=0;
18227|     WCHAR Name[255];
18228|     PNTFS_File File;
18229|
18230|     // name will be modified to contain uppcase of
    | passed in string
18231|     wcscpy(Name,FileName);
18232|     File = NTFS_OpenFileByName(VolumeInfo, Name,
    | FILENAME_ATTR, NULL );
18233|     if(File) {
18234|         // get data attribute
18235|         PATTRIBUTE Attr =
    | (PATTRIBUTE)NTFS_GetAttributeByName( File->VolumeInfo,
    | File->Mft, DATA_ATTR, NULL );
18236|         if(Attr) {
18237|             if(Attr->NotResident) {
18238|                 DLOG((TEXT("File = %s, mft
    | entry=%l64d\n"),FileName,File->MftEntryNum));
18239|                 DLOG((TEXT(" Allocated =
    | %12l64d\n"),Attr->NonResidentData.AllocatedDiskSpace));
18240|                 DLOG((TEXT(" File Size =
    | %12l64d\n"),Attr->NonResidentData.AttributeSize));
18241|                 DLOG((TEXT(" Initialized =
    | %12l64d\n"),Attr->NonResidentData.LengthOfInitializedDat
    | a));
18242|                 SAFE_ReadWrite(Attr);
18243|
    | Attr->NonResidentData.LengthOfInitializedData =
    | Attr->NonResidentData.AllocatedDiskSpace;
18244|                 SAFE_ReadOnly(Attr);
18245|                 Err = NTFS_CommitFile(File);
18246|
18247|                 if(!Err) {
18248|                     DLOG((TEXT("File committed\n")));
18249|                 } else {
18250|                     DLOG((TEXT("Error %08x committing
    | file\n"),Err));
18251|                 }
18252|             } else {
18253|                 DLOG((TEXT("Data is resident, nothing
    | to do\n")));
18254|                 Err = 0;
18255|             }
18256|         } else {
18257|             DLOG((TEXT("No data attribute found\n")));
18258|             Err = ERROR_NO_ATTRIBUTE_FOUND;

```

```

18259|     }
18260|
18261|     NTFS_CloseFile(File);
18262| } else {
18263|     DLOG((TEXT("Unable to find file
    | '%s'\n"),FileName));
18264|     Err = ERROR_FILE_NOT_FOUND;
18265| }
18266| return Err;
18267| }
18268|
18269|
18270| ULONG NTFS_AllocAndInitFiles( WCHAR *NTDeviceName,
    | ULONG NumFiles, tNTFS_AllocFiles Files[] )
18271| {
18272|     ULONG Err=0;
18273|     tDASDHandle *VolumeHandle;
18274|     ULONG i;
18275|
18276|     __try {
18277|         DLOG((TEXT("Opening volume\n")));
18278|         VolumeHandle=DASD_LockVolume(NTDeviceName);
18279|         if(VolumeHandle) {
18280|             ULONG Dismount;
18281|
18282|             DLOG((TEXT("Allocing space on volume\n")));
18283|             i=0;
18284|             do {
18285|                 Err =
    | AllocSpace(NTDeviceName,Files[i].FileNameOnly,Files[i].F
    | ileSize);
18286|                 i++;
18287|             } while((Err==0) && (i<NumFiles));
18288|
18289|             if(!Err) {
18290|                 DLOG((TEXT("Dismounting volume
    | %08x\n"),VolumeHandle->hDevice));
18291|
    | Dismount=DismountVolume(VolumeHandle->hDevice);
18292|                 if(Dismount) {
18293|                     tDASDHandle
    | *VolumeHandle2=DASD_LockVolume(NTDeviceName);
18294|
18295|                     if(VolumeHandle2) {
18296|                         tVolumeInfo *VolumeInfo;
18297|                         DLOG((TEXT("Loading volume
    | information\n")));
18298|
18299|                         VolumeInfo=NTFS_OpenVolume(
    | VolumeHandle2, 0);

```



```

18300|             if(VolumeInfo) {
18301|                 DLOG((TEXT("Initializing
| space on volume\n")));
18302|                 i = 0;
18303|                 do {
18304|                     Err =
| InitializeSpace(VolumeInfo,Files[i].FileNameOnly,Files[i
| ].FileSize);
18305|                     i++;
18306|                 } while ((Err==0) &&
| (i<NumFiles));
18307|
18308|                 if(Err) {
18309|                     DLOG((TEXT("Error %08x
| initing space for cache\n"),Err));
18310|                 }
18311|
| NTFS_CloseVolume(VolumeInfo);
18312|                 } else {
18313|                     DLOG((TEXT("Unable to open
| volume\n")));
18314|                     Err =
| ERROR_UNRECOGNIZED_VOLUME;
18315|                 }
18316|                 } else {
18317|                     DLOG((TEXT("Unable to open
| volume from os\n")));
18318|                     Err =
| ERROR_UNRECOGNIZED_VOLUME;
18319|                 }
18320|
18321|                 DASD_UnlockDevice(VolumeHandle2);
18322|                 } else {
18323|                     DLOG((TEXT("Error dismounting
| volume\n")));
18324|                     Err = ERROR_UNABLE_TO_UNLOAD_MEDIA;
18325|                 }
18326|                 } else {
18327|                     DLOG((TEXT("Error %08x allocating
| space\n"),Err));
18328|                 }
18329|                 DASD_UnlockDevice(VolumeHandle);
18330|                 } else {
18331|                     DLOG((TEXT("Error getting volume
| handle\n")));
18332|                     Err = ERROR_UNRECOGNIZED_VOLUME;
18333|                 }
18334|             } __except
| (ExceptionFilter(GetExceptionInformation())) {
18335|                 Err = GetExceptionCode();

```

```

18336|     DLOG((TEXT("NTFS_AllocAndInitFiles: Exception
      | %08x allocating and initing files\n"),Err));
18337| }
18338|
18339| // if any errors, go ahead and delete all the files
18340| // so PSM doesnt accidentally use the files when they
      | havent been
18341| // initialized or it will deadlock with itself
18342| if(Err) {
18343|     __try {
18344|         WCHAR FullName[256];
18345|
18346|         i=0;
18347|         do {
18348|             swprintf(FullName,
      | L"%s%s",NTDeviceName,Files[i].FileNameOnly);
18349|             DLOG((TEXT("Deleting file
      | '%s'\n"),FullName));
18350|             NtDeleteFile(FullName);
18351|             i++;
18352|         } while(i<NumFiles);
18353|     } __except
      | (ExceptionFilter(GetExceptionInformation())) {
18354|         Err = GetExceptionCode();
18355|         DLOG((TEXT("NTFS_AllocAndInitFiles:
      | Exception %08x deleting files\n"),Err));
18356|     }
18357|
18358| }
18359|
18360| return Err;
18361| }
18362|
18363|
18364|
18365| File Listing: ntfs.h
18366|
18367| /*
18368| Copyright 1997-98 Columbia Data Products, Inc.
      | All Rights Reserved!
18369|
18370| This file is a compilation of different resources:
18371|
18372| Helen Custer's "Inside the Windows NT File System"
18373|         Microsoft Press
18374|         ISBN: 1-55615-660-x
18375| Martin von Lewis and Regis Duchesne from the ntfs
      | linux driver
18376| Docs at
      | http://Celine.via.ecp.fr/~hpreg/ntfs/new/

```

```

18377|      Driver at
      | http://www.informatik.hu-berlin.de/~loewis/ntfs/
18378|      I have included comments from their docs at
      | some places.
18379|      Personal Note: The source code is very bad,
      | it makes no use of structures, so
18380|      except for reference, his code
      | is pretty much useless. IOW: I look
18381|      to see how he decompresses a
      | run, then implement my own.
18382|      Rob Green (me!) Columbia Data Products, Inc.
      | 407-869-6700
18383|      Reversed engineered Ntfs Boot Sector, and other
      | fields that Martin left out. Also
18384|      added information that i happen to know about
      | due to me reverse engineering other
18385|      file systems
18386|      Mark Russinovich and Bryce Cogswell's NTFSDos
      | Driver and NTFSInfo.
18387|      While no source code, these files were helpful
      | to verify results with
18388|      MS NT Resource Kit DskProbe.exe
18389|      Used to get sectors so i could reverse engineer
      | them. As a side note, while not
18390|      a bad disk editor, it could use some updating.
18391|      MS ntfs.sys driver
18392|      Some structures came from my endeavors of
      | reverse engineering the ntfs.sys driver
18393|      that comes included with Windows NT.
18394|
18395| */
18396|
18397|
18398| //-----
      | -----
18399|
18400| // converts a cluster number to the physical sector
      | number
18401| #define NTFS_ClusterToPhysical(VolumeInfo,ClusterNum)
      | (((VolumeInfo)->NtfsBootSector->SectorsPerCluster*(Clust
      | erNum))+((VolumeInfo)->VolumeStartSector))
18402| #define NTFS_ClusterToLogical(VolumeInfo,ClusterNum)
      | (((VolumeInfo)->NtfsBootSector->SectorsPerCluster*(Clust
      | erNum)))
18403| #define
      | NTFS_PhysicalToLogical(VolumeInfo,Physical,Logical)
      | ((Physical)-((VolumeInfo)->VolumeStartSector))
18404| #define
      | NTFS_LogicalToPhysical(VolumeInfo,Logical,Physical)
      | ((Logical)+((VolumeInfo)->VolumeStartSector))

```

```

18405|
18406| // gets a pointer into the data area of the attribute
18407| #define GetDataPointer(Attribute)
    | ((Attribute)->Offset ?
    | ((PVOID)&(((CHAR*)Attribute)[(Attribute)->Offset+((Attri
    | bute)->NameLength*2]))) : ((Attribute)->NotResident ?
    | ((PVOID)&(((CHAR*)Attribute)[(Attribute)->NonResidentDat
    | a.Offset+((Attribute)->NameLength*2)])) :
    | ((PVOID)&(((CHAR*)Attribute)[(Attribute)->ResidentData.O
    | ffset+((Attribute)->NameLength*2)])))
18408| #define NTFS_GetDataPointer(Attribute)
    | ((Attribute)->NotResident ?
    | ((PVOID)&(((CHAR*)Attribute)[(Attribute)->NonResidentDat
    | a.Offset])) :
    | ((PVOID)&(((CHAR*)Attribute)[(Attribute)->ResidentData.O
    | ffset]))))
18409| // retrieves the name of an attribute if it has one
18410| #define NTFS_GetAttributeNamePointer(Attribute)
    | ((Attribute)->NameLength ? ((Attribute)->Offset ?
    | ((PVOID)&(((CHAR*)Attribute)[(Attribute)->Offset])) :
    | ((Attribute)->NotResident ?
    | ((PVOID)&(((CHAR*)Attribute)[(Attribute)->NonResidentDat
    | a.Offset])) :
    | ((PVOID)&(((CHAR*)Attribute)[(Attribute)->ResidentData.O
    | ffset])))) : NULL)
18411| // size of the attribute
18412| #define NTFS_GetDataSize(Attribute)
    | ((Attribute)->NotResident ?
    | ((Attribute)->NonResidentData.AttributeSize.QuadPart) :
    | ((Attribute)->ResidentData.DataLength))
18413|
18414| // returns the size of the mft in bytes
18415| #define NTFS_MftByteSize(VolumeInfo)
    | ((VolumeInfo)->NtfsBootSector->MFTRecordSize*(VolumeInfo
    | )->NtfsBootSector->BytesPerSector)
18416| // returns the size of the cluster in bytes
18417| #define NTFS_ClusterSizeInBytes(VolumeInfo)
    | ((VolumeInfo)->NtfsBootSector->SectorsPerCluster*(Volume
    | Info)->NtfsBootSector->BytesPerSector)
18418| // size of an index entry for directory parsing
18419| #define NTFS_IndexEntrySize(IE)
    | ((IE)->EntrySize)
18420| // Gets VCN of subnodes for greater than cases during
    | directory parsing
18421| #define GetSubNodesVCN(IE)
    | ((ULONGLONG)*(&(((BYTE*)IE)[(IE)->EntrySize-8])))
18422| #define NTFS_GetSubNodesVCN(IE)
    | *((ULONGLONG*)(((BYTE*)IE)+(IE)->EntrySize-8))
18423|
18424|

```

18425|

18426|

18427|

18428| /\* MFT File Entry

18429| Interest

18430|

18431| This is a component of the MFT. Each file of the volume  
| is completely described by

18432| some of these FILE records. The first FILE record that  
| describes a given file is

18433| called the base FILE record (or an inode in Linux  
| terminology). All other are called

18434| extension FILE records.

18435|

18436| Layout

18437|

18438| A FILE record is 1 KB large or the cluster size if  
| larger. It falls into 2 parts :

18439|

18440|     Flags

18441|         Bit     Signification

18442|         01     Non-resident attributes

18443|         02     The FILE record describes a  
| directory

18444|

18445|

18446|     base FILE record number

18447|         If the FILE record is a base FILE  
| record, the number is 0000000000000000.

18448|         If the FILE record is an extension FILE  
| record, the number is its

18449|         corresponding base FILE record number.

18450|

18451|     The sequence of attributes part

18452|

18453|     This is a sequence of file attributes that has a  
| variable length. In each FILE record,

18454|     the sequence is ordered by increasing order of the  
| attribute type. The sequence is

18455|     terminated with FFFFFFFF.

18456|

18457| Properties

18458|

18459| Extension FILE records are used when all information  
| about a file doesn't fit into the

18460| base FILE record (e.g. if the sequence of file  
| attributes grows). Only the base FILE

18461| record is used for referencing the file it describes.  
| Since the type of the Attribute List

18462| file attribute is small enough, we are sure that this

```

| file attribute will be in the base
18463| FILE record. And this file attribute provides the
| references to all the extension FILE
18464| records describing the file.
18465|
18466| Questions
18467|
18468| What does the flag 'Non-resident attributes' in the
| header part mean? Is it set only when
18469| all file attributes in the sequence of attributes part
| are non-resident?
18470| */
18471|
18472| #define MFT_FLAG_NOT_RESIDENT 0x0001
18473| #define MFT_FLAG_DIRECTORY 0x0002
18474|
18475| #pragma pack(1)
18476|
18477| // first mft(which holds all mfts) is stored at
| NtfsBootSector->MftCluster
18478| typedef struct _MFT {
18479|     ULONG Signature; // FILE
18480|     WORD FixupOffset;
18481|     WORD NumberOfFixups;
18482|     ULARGE_INTEGER Unknown;
18483|     WORD SequenceNumber;
18484|     WORD HardLinkCount;
18485|     WORD OffsetToAttributes;
18486|     WORD Flags;
18487|     ULONG LengthInUse;
18488|     ULONG Allocated;
18489|     ULARGE_INTEGER MftRecord;
18490|     WORD MaxAttributeNumber;
18491|     WORD Fixups[(0x400-0x2a) / 2]; // only 1k for
| default...
18492|     // fixups follow...
18493| } MFT, *PMFT;
18494|
18495| // Standard attributes
18496| #define STANDARD_INFORMATION_ATTR 0x10
18497| #define ATTRIBUTE_LIST_ATTR 0x20
18498| #define FILENAME_ATTR 0x30
18499| #define VOLUME_VERSION_ATTR 0x40
18500| #define SECURITY_DESCRIPTOR_ATTR 0x50
18501| #define VOLUME_NAME_ATTR 0x60
18502| #define VOLUME_INFO_ATTR 0x70
18503| #define DATA_ATTR 0x80
18504| #define INDEX_ROOT_ATTR 0x90
18505| #define INDEX_ALLOCATION_ATTR 0xa0
18506| #define BITMAP_ATTR 0xb0

```

```

18507| #define SYMLINK_ATTR          0xc0
18508| #define HPFS_EA_INFO_ATTR      0xd0
18509| #define HPFS_EA_ATTR          0xe0
18510|
18511|
18512| typedef struct _RESIDENT_ATTR {
18513|     ULONG   DataLength;           // 10
18514|     WORD    Offset;               // 14
18515|     WORD    AttributesIndexed;    // 16
18516| // BYTE    Data[1];              // 18
18517| } RESIDENT_ATTR;
18518|
18519| typedef struct _NONRESIDENT_ATTR {
18520|     ULARGE_INTEGER StartingVCN;    //
18521|     | 10
18522|     ULARGE_INTEGER LastVCN;       //
18523|     | 18
18524|     WORD          Offset;         //
18525|     | 20
18526|     WORD          CompressionEngine; //
18527|     | 22
18528|     ULONG         Unknown;        //
18529|     | 24
18530|     ULARGE_INTEGER AllocatedDiskSpace; //
18531|     | 28
18532|     ULARGE_INTEGER AttributeSize;  //
18533|     | 30
18534|     ULARGE_INTEGER LengthOfInitializedData; //
18535|     | 38
18536|     ULARGE_INTEGER CompressedSize;  //
18537|     | 40
18538| // BYTE          Data[1];          //
18539|     | 48
18540| } NONRESIDENT_ATTR;
18541|
18542| typedef struct _ATTRIBUTE {
18543|     ULONG   Type;                 // 0
18544|     ULONG   Length;               // 4
18545|     BYTE    NotResident;          // 8
18546|     BYTE    NameLength;           // 9
18547|     BYTE    Offset;               // a
18548|     BYTE    Unknown1;             // b
18549|     BYTE    Compressed;           // c
18550|     BYTE    Unknown2;             // d
18551|     WORD    AttributeId;          // e
18552|     union {
18553|         RESIDENT_ATTR ResidentData; // 10
18554|         NONRESIDENT_ATTR NonResidentData;
18555|     };
18556| } ATTRIBUTE, *PATTRIBUTE; // Resident size = 18,

```

```

| NonResident size = 48
18547|
18548|
18549| #define DOS_READONLY    0x0001
18550| #define DOS_HIDDEN      0x0002
18551| #define DOS_SYSTEM       0x0004
18552| #define DOS_ARCHIVE      0x0020
18553| #define DOS_COMPRESSED   0x0800
18554|
18555| typedef struct _STANDARD_INFORMATION {
18556|     ULARGE_INTEGER FileCreationTime;    // 18
18557|     ULARGE_INTEGER LastModifiedTime;    // 20
18558|     ULARGE_INTEGER LastModifiedTimeForMFT; // 28
18559|     ULARGE_INTEGER LastAccessTime;      // 30
18560|     ULONG          DosAttributes;       // 38
18561|     BYTE           Unused[0xc];         // 3c
18562| } STANDARD_INFORMATION, *PSTANDARD_INFORMATION;
18563|
18564| typedef struct _ATTRIBUTE_LIST {
18565|     ULONG          Type;                // 0
18566|     WORD           RecordLength;        // 4
18567|     BYTE           NameLength;          // 6
18568|     BYTE           Unknown1;            // 7
18569|     ULARGE_INTEGER StartingVCN;         // 8
18570|     ULONGLONG      MFTEntry              : 48; // 10
18571|     ULONGLONG      SequenceNumber        : 16; // 16
18572|     WORD           AttributeID;          // 18
18573|     WORD           Unknown2;             // 1a
18574|     WORD           Unknown3;             // 1c
18575|     WORD           Unknown4;             // 1e
18576|     // WCHAR       Name[1];              // 20
18577| } ATTRIBUTE_LIST, *PATTRIBUTE_LIST;
18578|
18579| #define FILENAME_POSIX      0x0
18580| #define FILENAME_UNICODE    0x1
18581| #define FILENAME_DOS        0x2
18582| #define FILENAME_UNICODE_DOS 0x3
18583|
18584| #define FILENAME_READONLY    0x00000001
18585| #define FILENAME_HIDDEN      0x00000002
18586| #define FILENAME_SYSTEM      0x00000004
18587| #define FILENAME_ARCHIVE     0x00000020
18588| #define FILENAME_COMPRESSED  0x00000800
18589| #define FILENAME_DIRECTORY   0x10000000
18590|
18591| typedef struct _FILENAME {
18592|     ULONGLONG      DirectoryMft          : 48; //
| 18
18593|     ULONGLONG      SequenceNumber        : 16;
18594|     ULARGE_INTEGER FileCreationTime;      //

```



```

| 20
18595|  ULARGE_INTEGER LastModifiedTime;          //
| 28
18596|  ULARGE_INTEGER LastModifiedTimeForMFT;      //
| 30
18597|  ULARGE_INTEGER LastAccessTime;              //
| 38
18598|  ULARGE_INTEGER FileSize;                    //
| 40
18599|  ULARGE_INTEGER AttributeSize;               //
| 48
18600|  ULONG          Flags;                        //
| 50
18601|  ULONG          Unknown;                      //
| 54
18602|  BYTE          FileNameLength;                //
| 58
18603|  BYTE          FileNameType;                  //
| 59
18604|  WCHAR          FileName[1];                 //
| 5a
18605| } FILENAME, *PFILENAME;
18606|
18607| // The first 11 inodes correspond to special files
18608| #define FILE_MFT      0
18609| #define FILE_MFTMIRR  1
18610| #define FILE_LOGFILE  2
18611| #define FILE_VOLUME   3
18612| #define FILE_ATTRDEF  4
18613| #define FILE_ROOT     5
18614| #define FILE_BITMAP   6
18615| #define FILE_BOOT     7
18616| #define FILE_BADCLUS  8
18617| #define FILE_QUOTA    9
18618| #define FILE_UPCASE  10
18619|
18620| /*
18621|  Files 11..15 are reserved also. 15 is marked as
  | 'BAAD' instead of 'FILE' for some
18622|  reason...
18623| */
18624|
18625| typedef struct _VOLUME_INFORMATION {
18626|  BYTE Unknown[8];
18627|  BYTE Unknown2;          // 1 on my drive
18628|  BYTE Unknown3;          // 2 on my drive
  | (maybe a version?)
18629|  BYTE Chkdsk;            // 1 = volume will be
  | chkdsk /f during boot
18630|  BYTE Unknown4[5];

```

```

18631| } VOLUME_INFORMATION, *PVOLUME_INFORMATION;
18632|
18633| #define ATTR_FLAG_INDEXABLE    0x0000000000000001
18634| #define ATTR_FLAG_REGENERATE    0x0000000000000040 //
    | regenerate before the regeneration phase
18635| #define ATTR_FLAG_NON_RESIDENT  0x0000000000000080 //
    | not sure on this one
18636|
18637| typedef struct _ATTRIBUTE_DEFINITION {
18638|     WCHAR      Name[40];
18639|     ULARGE_INTEGER Type;
18640|     ULARGE_INTEGER Flags;
18641|     ULARGE_INTEGER MinimumSize;
18642|     ULARGE_INTEGER MaximumSize;
18643| } ATTRIBUTE_DEFINITION, *PATTRIBUTE_DEFINITION;
18644|
18645| typedef struct _UPCASE {
18646|     WCHAR UpperCaseCharacters[65536];
18647| } UPCASE, *PUPCASE;
18648|
18649| typedef struct _VOLUME_NAME {
18650|     WCHAR VolumeName[1]; // actually counted unicode
    | string, count is in Attribute->ResidentData.DataLength
18651| } VOLUME_NAME, *PVOLUME_NAME;
18652|
18653| #define FLAGS_INDEX_SUBNODES    0x00000001
18654| #define FLAGS_INDEX_LAST        0x00000002
18655|
18656| typedef struct _INDEX_ENTRY {
18657|     ULONGLONG    DirectoryMft      : 48;    //
    | 0
18658|     ULONGLONG    SequenceNumber    : 16;
18659|     WORD          EntrySize;        //
    | 8
18660|     // Still not sure on this one...
18661|     WORD          DataSize;         //
    | a
18662|     ULONG         Flags;            //
    | c
18663|     ULONGLONG    MyDirectoryEntry  : 48;    //
    | 10
18664|     ULONGLONG    MySequenceNumber  : 16;
18665| #if 0
18666|     ULARGE_INTEGER FileCreationTime;        //
    | 18
18667|     ULARGE_INTEGER LastModifiedTime;        //
    | 20
18668|     ULARGE_INTEGER LastModifiedTimeForMFT;   //
    | 28
18669|     ULARGE_INTEGER LastAccessTime;          //

```

```

| 30
18670| #else
18671| FILETIME FileCreationTime; //
| 18
18672| FILETIME LastModifiedTime; //
| 20
18673| ULARGE_INTEGER LastModifiedTimeForMFT; //
| 28
18674| FILETIME LastAccessTime; //
| 30
18675| #endif
18676| ULARGE_INTEGER AllocatedAttributeSize; //
| 38
18677| ULARGE_INTEGER AttributeSize; //
| 40
18678| ULARGE_INTEGER Attributes; //
| 48 , FILENAME_DIRECTORY, Etc...
18679| // since this is an index, this may be KeyLength,
| KeyType, and KeyData
18680| BYTE FileNameLength; //
| 50
18681| BYTE FileNameType; //
| 51
18682| WCHAR FileName[1]; //
| 52
18683| // EntrySize-8 = VCN of index buffer with
| subnodes
18684| } INDEX_ENTRY, *PINDEX_ENTRY;
18685|
18686| typedef struct _INDEX_ALLOCATION {
18687| ULONG Signature; //
| 0
18688| WORD OffsetToFixup; //
| 4
18689| WORD NumberOfFixups; //
| 6
18690| ULARGE_INTEGER Unknown; //
| 8
18691| ULARGE_INTEGER VCNOOfBuffer; //
| 10 Back pointer to self
18692| WORD HeaderSize; //
| 18
18693| WORD Unknown2; //
| 1a
18694| ULONG Length; //
| 1c
18695| ULONG AllocatedLength; //
| 20
18696| ULONG Unknown3; //
| 24

```

```

18697|  WORD      Fixup;           //
      | 28
18698| } INDEX_ALLOCATION, *PINDEX_ALLOCATION;
18699|
18700| typedef struct _INDEX_ROOT {
18701|  ULONG      Signature;
18702|  ULONG      Unknown1;
18703|  ULONG      SizeOfIndex;
18704|  ULONG      NumberOfClustersPerIndex;
18705|  ULONG      Unknown2;
18706|  ULONG      Unknown3;
18707|  ULONG      Unknown4;
18708|  ULONG      Unknown5;
18709| } INDEX_ROOT, *PINDEX_ROOT;
18710|
18711| #pragma pack()
18712|
18713| // my structure!
18714|
18715| typedef struct _DATA_RUN {
18716|  ULONGLONG  Cluster;
18717|  ULONGLONG  Length;
18718| } DATA_RUN, *PDATA_RUN;
18719|
18720| typedef struct sFragmentDescriptor {
18721|  DATA_RUN  dr;
18722|  LIST_ENTRY Link;
18723| } tFragmentDescriptor, *pFragmentDescriptor;
18724|
18725| typedef struct _FAT16 {
18726|  WORD  Cluster;
18727| } FAT16, *PFAT16;
18728|
18729| typedef struct _FAT32 {
18730|  ULONG  Cluster;
18731| } FAT32, *PFAT32;
18732|
18733| typedef struct _FAT12 {
18734|  WORD  Cluster; // only 12 bits per entry!!!
18735| } FAT12, *PFAT12;
18736|
18737| typedef struct _VolumeInfo {
18738|  pDASDHandle  LockHandle;
18739|  ULONGLONG    VolumeStartSector;
18740|
18741|  // ntfs stuff
18742|  PNTFS_BOOT_SECTOR  NtfsBootSector;
18743|  PMFT                Mft;
18744|  struct _NTFS_File  *MftFile;
18745|  WCHAR              *UppcaseTable;

```

```

18746|
18747| #if 0
18748|     // fat stuff
18749|     PFAT_BOOT_SECTOR    FatBootSector;
18750|     PFAT16               Fat;
18751|     PVOID                RootDir;
18752| #endif
18753| } tVolumeInfo, *pVolumeInfo;
18754|
18755| typedef struct _NTFS_File {
18756|     pVolumeInfo          VolumeInfo;
18757|     ULONG                Type;
18758|     WCHAR                *AttrName;
18759|     ULONG                VCNTToLCNMappingTableSize;
18760|     PDATA_RUN            VCNTToLCNMappingTable;
18761|     ULONGLONG            MftEntryNum;
18762|     PMFT                 Mft;
18763|     PVOID                Buffer;
18764|     PBYTE                CompressedBuffer;
18765|     PBYTE                UnCompressedBuffer;
18766| } NTFS_File, *PNTFS_File;
18767|
18768|
18769| // NTFS support functions
18770| int GetRun ( PVOID DataRun, ULONG *Offset,
18771|             | ULARGE_INTEGER *Cluster, ULARGE_INTEGER *Length, ULONG
18772|             | *Sparse );
18773| PVOID NTFS_GetDataRun(PATTRIBUTE Attribute, PNTFS_File
18774|             | NtfsFile, pVolumeInfo VolumeInfo, ULONGLONG
18775|             | MftEntryNum, ULONG Type, WCHAR *AttrName );
18776| ULONGLONG AdjustGapToRelative( PVOID DataRunComplex,
18777|             | PVOID DataRun, PVOID DataRunComplexLimit);
18778| ULONGLONG FindClusterForEntryNum( PVOID DataRun,
18779|             | ULONGLONG EntryNum);
18780| ULONG GetDataRunLength( PVOID DataRun, ULONGLONG
18781|             | NumClusters );
18782| ULARGE_INTEGER GetPackedDataRunFinalStartLCN( PVOID
18783|             | DataRun, PVOID DataRunComplexLimit);
18784| ULONG GetPackedDataRunLength( PVOID DataRun, ULONGLONG
18785|             | NumClusters );
18786| ULONG MakeVCNTToLCNMapping ( PVOID DataRun, PDATA_RUN
18787|             | MappingTable, ULONG NumRuns );
18788| ULONGLONG CountClearBits( PVOID Data, ULONGLONG
18789|             | ByteCount );
18790| ULONGLONG CountSetBits( PVOID Data, ULONGLONG ByteCount
18791|             | );
18792|
18793| // ntfs low level functions
18794| int NTFS_FixupRecord ( pVolumeInfo VolumeInfo,
18795|             | PVOID Record );

```

```

18783| ULONGLONG NTFS_VCNToLCN(PNTFS_File File, ULONGLONG VCN
    | );
18784| ULONG NTFS_ReadLogicalCluster( pVolumeInfo
    | VolumeInfo, ULONGLONG Cluster, ULONG NumClusters, PVOID
    | Buffer );
18785| ULONG NTFS_WriteLogicalCluster( pVolumeInfo
    | VolumeInfo, ULONGLONG Cluster, ULONG NumClusters, PVOID
    | Buffer );
18786| PVOID NTFS_GetAttributeByType( pVolumeInfo
    | VolumeInfo, PMFT Mft, ULONG Type );
18787| PVOID NTFS_GetAttributeByName( pVolumeInfo
    | VolumeInfo, PMFT Mft, ULONG Type, WCHAR *Name );
18788| ULONG NTFS_ReadMftEntryNumber( pVolumeInfo
    | VolumeInfo, ULONGLONG EntryNum, PVOID Buffer );
18789| ULONG NTFS_WriteMftEntryNumber( pVolumeInfo
    | VolumeInfo, ULONGLONG EntryNum, PVOID Buffer );
18790|
18791|
18792| ULONGLONG DisplayFilesInDir( tVolumeInfo *VolumeInfo,
    | ULONGLONG MftEntry );
18793|
18794| //ray, check this has to be above first use. When
    | proved find its final resting place
18795| typedef struct _NODE {
18796|     PVOID ShallowerNode;
18797|     PINDEX_ALLOCATION IA;
18798|     PINDEX_ENTRY IE;
18799| } tNODE,*PNODE;
18800| typedef struct _NTFS_FIND_INFO {
18801|     PNTFS_File DirectoryFile;
18802|     PNTFS_File IRFile;
18803|     PNTFS_File IAFile;
18804|     LONGLONG IRSize;
18805|     PNODE DeepestNode;
18806|     PINDEX_ROOT IR;
18807|     PINDEX_ENTRY IE;
18808| } tNTFS_FIND_INFO,*PNTFS_FIND_INFO;
18809|
18810| typedef struct _NTFS_FIDO_INFO {
18811|     PNTFS_File DirectoryFile;
18812|     ULONGLONG IndexesFound;
18813| } tNTFS_FIDO_INFO,*PNTFS_FIDO_INFO;
18814|
18815|
18816| // ntfs high level functions
18817| pVolumeInfo NTFS_OpenVolume( tDASDHandle *LockHandle,
    | ULONGLONG Sector );
18818| ULONGLONG NTFS_GetNumFreeClusters( pVolumeInfo
    | VolumeInfo );
18819| PNTFS_File NTFS_OpenFileByName ( pVolumeInfo

```

```

    | VolumeInfo, WCHAR *Name, ULONG Type, WCHAR *AttrName );
18820| PNTFS_File NTFS_OpenFileByNumber ( pVolumeInfo
    | VolumeInfo, ULONGLONG MftEntryNum, ULONG Type, WCHAR
    | *AttrName );
18821| ULONGLONG NTFS_GetFileSize ( PNTFS_File File );
18822| ULONGLONG NTFS_GetCompressedFileSize( PNTFS_File File
    | );
18823| ULONG NTFS_GetFileInfo( PNTFS_File File,
    | PFILENAME FN );
18824| ULONG NTFS_ReadFile ( PNTFS_File File,
18825| ULONGLONG ByteOffset,
18826| ULONGLONG ByteCount,
18827| PVOID Data );
18828| ULONG NTFS_WriteFile ( PNTFS_File File,
18829| ULONGLONG ByteOffset,
18830| ULONGLONG ByteCount,
18831| PVOID Data );
18832| ULONG NTFS_CommitFile ( PNTFS_File File );
18833| ULONG NTFS_CloseFile( PNTFS_File File );
18834| void NTFS_CloseVolume( pVolumeInfo VolumeInfo );
18835| BOOL NTFS_FindNextFile( PNTFS_FIND_INFO hFindFile,
    | LPWIN32_FIND_DATA lpFindFileData );
18836| BOOL NTFS_FidoNextFile( PNTFS_FIDO_INFO hFindFile,
    | LPWIN32_FIND_DATA lpFindFileData );
18837|
18838| // psuedo functions
18839| #define NTFS_GetVolumeSizeInClusters(VolumeInfo)
    | ((VolumeInfo)->NtfsBootSector->NumberOfSectors.QuadPart
    | / (VolumeInfo)->NtfsBootSector->SectorsPerCluster)
18840| #define NTFS_GetVolumeSizeInSectors(VolumeInfo)
    | ((VolumeInfo)->NtfsBootSector->NumberOfSectors)
18841| #define NTFS_GetFileNumber(File)
    | ((File)->MftEntryNum)
18842|
18843| // My error codes
18844|
18845| #define ERROR_NO_ATTRIBUTE_FOUND 0xe8000001
18846| #define ERROR_INVALID_RUN 0xe8000002
18847| #define ERROR_NOT_COMPRESSED 0xe8000003
18848|
18849|
18850| typedef struct sNTFS_AllocFiles {
18851| WCHAR *FileNameOnly;
18852| LARGE_INTEGER FileSize;
18853| } tNTFS_AllocFiles, *pNTFS_AllocFiles;
18854|
18855| ULONG NTFS_AllocAndInitFiles( WCHAR *NTDeviceName,
    | ULONG NumFiles, tNTFS_AllocFiles Files[] );
18856|
18857|

```

```

18858|
18859| File Listing: ntfs_ondisk.h
18860|
18861| #pragma pack(1)
18862| typedef struct _NTFS_BOOT_SECTOR {
18863|     BYTE    Jmp[3];
18864|     BYTE    OemId[8];
18865|     USHORT  BytesPerSector;
18866|     BYTE    SectorsPerCluster;
18867|     USHORT  ReservedSectors;    // not used in NT
18868|     | (0)
18869|     BYTE    NumberOfFats;    // not used in NT
18870|     | (0)
18871|     USHORT  RootDirectory;    // not used in NT
18872|     | (0)
18873|     USHORT  SmallSectors;    // not used in NT
18874|     | (0)
18875|     BYTE    MediaId;
18876|     USHORT  SectorsPerFat;    // not used in NT
18877|     | (0)
18878|     USHORT  SectorsPerTrack;
18879|     USHORT  Heads;
18880|     ULONG   HiddenSectors;    // not used in NT
18881|     | (0)
18882|     ULONG   LargeSectors;    // not used in NT
18883|     | (0)
18884|     BYTE    Drive;
18885|     BYTE    DirtyVolume;
18886|
18887|     USHORT  Reserved;
18888|     ULARGE_INTEGER NumberOfSectors;
18889|     ULARGE_INTEGER MftCluster;
18890|     ULARGE_INTEGER Mft2Cluster;
18891|
18892|     ULONG   MFTRecordSize;
18893|     ULONG   IndexBufferSize;
18894|     /* conflicting stories....
18895|     BYTE    MFTRecordSize;
18896|     ULONG   IndexBufferSize;
18897|     USHORT  Unknown1;
18898|     BYTE    Unknown2;
18899|
18900|     */
18901|     ULARGE_INTEGER SerialNumber;
18902|     ULONG   SectorChecksum;
18903|     ULONG   Unknown1;
18904|     ULONG   Unknown2;
18905|     BYTE    Unknown3;
18906|     BYTE    Code[0x1a1];
18907|     USHORT  Signature; // aa55

```



```

18901| } NTFS_BOOT_SECTOR;
18902| typedef NTFS_BOOT_SECTOR *PNTFS_BOOT_SECTOR;
18903| #pragma pack()
18904|
18905|
18906|
18907| File Listing: ondisk.h
18908|
18909| /* general partition stuff */
18910|
18911| #define PARTITION_Empty          0x00
18912| #define PARTITION_DOS12Bit      0x01 // 12 bit fat
18913| | < 10mb
18914| #define PARTITION_DOS16Bit      0x04 // 16 bit fat
18915| | < 32mb
18916| #define PARTITION_DOSExtended   0x05 // Extended
18917| | Dos Partition
18918| #define PARTITION_DOSHuge       0x06 // 16 bit FAT
18919| | >= 32mb
18920| #define PARTITION_NTFS          0x07
18921|
18922| #define PARTITION_DRDOSHuge     0xC4
18923| #define PARTITION_DRDOSExtended 0xC5
18924| #define PARTITION_DRDOSCompress 0xC6
18925|
18926| #define PARTITION_ConCurDOS    0xDB
18927|
18928| #define PARTITION_Netware286    0x64
18929| #define PARTITION_Netware386    0x65
18930|
18931| #define GetBeginSPT(Part) ((Part)->BeginSector & 0x3f)
18932| #define GetEndSPT(Part) ((Part)->EndSector & 0x3f)
18933| #define GetBeginCyl(Part) (((Part)->BeginSector &
18934| | 0xc0) << 2) + (Part)->BeginCyl)
18935| #define GetEndCyl(Part) (((Part)->EndSector & 0xc0) <<
18936| | 2) + (Part)->EndCyl)
18937|
18938| #pragma pack(1)
18939| typedef struct _PartInfo {
18940|     BYTE    Bootable;
18941|     BYTE    BeginHead;
18942|     BYTE    BeginSector;
18943|     BYTE    BeginCyl;
18944|     BYTE    SystemType;
18945|     BYTE    EndHead;
18946|     BYTE    EndSector;
18947|     BYTE    EndCyl;
18948|     ULONG   StartSector;
18949|     ULONG   NumberOfSectors;

```

```

18945| } tPartInfo;
18946|
18947| #pragma pack(1)
18948| // physical sector 0
18949| typedef struct _MBR {
18950|     BYTE    Code[0x1b8];
18951|     ULONG    SerialNumber;
18952|     BYTE    Dirty;
18953|     BYTE    Unknown;
18954|     tPartInfo Part[4];
18955|     USHORT    Signature;
18956| } tMBR, *pMBR;
18957|
18958| #pragma pack(1)
18959| typedef struct _NTFS_BOOT_SECTOR {
18960|     BYTE    Jmp[3];
18961|     BYTE    OemId[8];
18962|     USHORT    BytesPerSector;
18963|     BYTE    SectorsPerCluster;
18964|     USHORT    ReservedSectors;    // not used in NT
18965|     | (0)
18966|     BYTE    NumberOfFats;    // not used in NT
18967|     | (0)
18968|     USHORT    RootDirectory;    // not used in NT
18969|     | (0)
18970|     USHORT    SmallSectors;    // not used in NT
18971|     | (0)
18972|     BYTE    MediaId;
18973|     USHORT    SectorsPerFat;    // not used in NT
18974|     | (0)
18975|     USHORT    SectorsPerTrack;
18976|     USHORT    Heads;
18977|     ULONG    HiddenSectors;    // not used in NT
18978|     | (0)
18979|     ULONG    LargeSectors;    // not used in NT
18980|     | (0)
18981|     BYTE    Drive;
18982|     BYTE    DirtyVolume;
18983|
18984|     USHORT    Reserved;
18985|     ULARGE_INTEGER NumberOfSectors;
18986|     ULARGE_INTEGER MftCluster;
18987|     ULARGE_INTEGER Mft2Cluster;
18988|
18989|     ULONG    MFTRecordSize;
18990|     ULONG    IndexBufferSize;
18991|     /* conflicting stories....
18992|     BYTE    MFTRecordSize;
18993|     ULONG    IndexBufferSize;
18994|     USHORT    Unknown1;

```

```

18988|  BYTE   Unknown2;
18989|
18990| */
18991|  ULARGE_INTEGER SerialNumber;
18992|  ULONG    SectorChecksum;
18993|  ULONG    Unknown1;
18994|  ULONG    Unknown2;
18995|  BYTE     Unknown3;
18996|  BYTE     Code[0x1a1];
18997|  USHORT   Signature; // aa55
18998| } NTFS_BOOT_SECTOR;
18999| typedef NTFS_BOOT_SECTOR *PNTFS_BOOT_SECTOR;
19000|
19001| #pragma pack(1)
19002| typedef struct _FAT_BOOT_SECTOR {
19003|  BYTE   Jmp[3];
19004|  BYTE   OemId[8];
19005|  USHORT BytesPerSector;
19006|  BYTE   SectorsPerCluster;
19007|  USHORT ReservedSectors;
19008|  BYTE   NumberOfFats;
19009|  USHORT RootDirectory;
19010|  USHORT SmallSectors;
19011|  BYTE   MediaId;
19012|  USHORT SectorsPerFat;
19013|  USHORT SectorsPerTrack;
19014|  USHORT Heads;
19015|  ULONG   HiddenSectors;
19016|  ULONG   LargeSectors;
19017|  BYTE    Drive;
19018|  BYTE    DirtyVolume;
19019|
19020|  BYTE    BootSignature; // 29 means the next
      | are valid
19021|  ULONG   VolumeID;
19022|  CHAR    VolumeLabel[0xb];
19023|  CHAR    FileSysType[8];
19024|  BYTE    Code[0x1c0];
19025|  USHORT   Signature; // aa55
19026| } FAT_BOOT_SECTOR;
19027| typedef FAT_BOOT_SECTOR *PFAT_BOOT_SECTOR;
19028| typedef FAT_BOOT_SECTOR GENERIC_BOOT_SECTOR;
19029| typedef GENERIC_BOOT_SECTOR *PGENERIC_BOOT_SECTOR;
19030|
19031| #pragma pack(1)
19032| typedef struct _FAT_DIR_ENTRY {
19033|  CHAR   Name[8];
19034|  CHAR   Ext[3];
19035|  BYTE   Attributes;
19036|  BYTE   Reserved[0xa];

```

```

19037|  USHORT  Time;
19038|  USHORT  Date;
19039|  USHORT  StartCluster;
19040|  ULONG   FileSize;
19041| } FAT_DIR_ENTRY;
19042| typedef FAT_DIR_ENTRY *PFAT_DIR_ENTRY;
19043|
19044| #pragma pack()
19045|
19046|
19047|
19048| File Listing: PHYSAPI.h
19049|
19050| PSMSTATUS PSMAPI Psm_Enable_PhysicalW( IN
    | pOpenTransactionInW In,
19051|                                     IN ULONG
    | NumDrives,
19052|                                     IN ULONG
    | *DriveMap );
19053| PSMSTATUS PSMAPI Psm_Enable_PhysicalExW( IN
    | pOpenTransactionInW In,
19054|                                     IN ULONG
    | NumDrives,
19055|                                     IN ULONG
    | *DriveMap,
19056|                                     INOUT
    | LPOVERLAPPED Overlapped,
19057|                                     IN
    | LPOVERLAPPED_COMPLETION_ROUTINE CompletionRoutine
19058|                                     );
19059| PSMSTATUS PSMAPI Psm_Enable_PhysicalA( IN
    | pOpenTransactionInA In,
19060|                                     IN ULONG
    | NumDrives,
19061|                                     IN ULONG
    | *DriveMap );
19062| PSMSTATUS PSMAPI Psm_Enable_PhysicalExA( IN
    | pOpenTransactionInA In,
19063|                                     IN ULONG
    | NumDrives,
19064|                                     IN ULONG
    | *DriveMap,
19065|                                     INOUT
    | LPOVERLAPPED Overlapped,
19066|                                     IN
    | LPOVERLAPPED_COMPLETION_ROUTINE CompletionRoutine
19067|                                     );
19068| #ifdef UNICODE
19069| #define Psm_Enable_Physical  Psm_Enable_PhysicalW
19070| #define Psm_Enable_PhysicalEx Psm_Enable_PhysicalExW

```

```

19071| #else
19072| #define Psm_Enable_Physical Psm_Enable_PhysicalA
19073| #define Psm_Enable_PhysicalEx Psm_Enable_PhysicalExA
19074| #endif
19075|
19076| PSMSTATUS PSMAPI Psm_Disable_Physical( );
19077| PSMSTATUS PSMAPI Psm_FreePhysicalDrives( IN ULONG
    | NumDrives,
19078|                                     IN ULONG
    | *Drives );
19079|
19080|
19081|
19082| File Listing: psmlapi.c
19083|
19084| #include <stdio.h>
19085| #include <stdlib.h>
19086| #include <string.h>
19087| #include <stddef.h>
19088| #include <windows.h>
19089| #include <tchar.h>
19090| #include <process.h>
19091| #include <time.h>
19092| #include <direct.h>
19093| #include <lm.h>
19094| #include <assert.h>
19095| #define ASSERT assert
19096|
19097| #include <winioctl.h>
19098| #include <undoc.h>
19099| // psm api
19100| #include <psm.h>
19101| // ioctls we need to send down
19102| #include "..\driver\ioctl.h"
19103|
19104| #include "volume.h"
19105|
19106| #include "defrag.h"
19107| #include <mountdev.h>
19108| #include <ntddstor.h>
19109| #include <ntddvol.h>
19110| #include <aclapi.h>
19111| #include <clusapi.h>
19112|
19113| #include "setup5.h"
19114| #include "setup4.h"
19115| #include "service.h"
19116| #include "cluster.h"
19117|
19118|

```

```

19119| // The following function resides in cacheloc.c
19120| extern ULONG FindBestVolumeForCache (
19121|     pOpenTransactionIn3W In,
19122|     WCHAR *TempPath,
19123|     int TempPathSizeInChars );
19124|
19125| #define DN_MakePointer(Start,Offset)
19126|     | ((PVOID)(((char*)(Start))+(Offset)))
19127| #define DN_MakeOffset(Start,Offset)
19128|     | ((ULONG)(((char*)(Offset))-((char*)(Start))))
19129| #define PSM_VOLUME_NAME_PATTERN
19130|     | L"\\?\\PsmVolume{%08x,%02x}"
19131| #define PSM_VOLUME_SHARE_PATTERN L"Psm%s_%d"
19132|
19133| #define try_return(S) { S; goto try_exit; }
19134|
19135| typedef struct sAPCContext {
19136|     tOpenTransactionIn3W In;
19137|     pOpenTransactionOutW Out;
19138|     pOpenTransactionInInternal OTI;
19139|     pOpenTransactionOutInternal OTO;
19140|     ULONG SizeofOTO;
19141|     ULONG SizeofOTI;
19142|     ULONG SizeOfVolumeMap;
19143|     PVOID VolumeMap;
19144|     ULONG *VolumeMapFlags;
19145|     IO_STATUS_BLOCK IoStatus;
19146|     pSnapShot *SnapShot;
19147|     tPSM_SecurityInfo SecurityInfo;
19148|     LPOVERLAPPED Overlapped;
19149|     LPOVERLAPPED_COMPLETION_ROUTINE OverlappedRoutine;
19150| } tAPCContext, *pAPCContext;
19151|
19152| // do not include psmlapi.h unless you undefine the
19153| // defines in it!
19154|
19155| #ifdef WIN32
19156| #define DLLEXPORT __declspec( dllexport )
19157| #define DLLIMPORT __declspec( dllimport )
19158| #endif
19159| #define UNREFERENCED(x) (x)=(x)
19160|
19161| #ifdef _DEBUG
19162| #define STATIC
19163| void PSM_LogDebugInfo( const TCHAR *fmt,...);
19164| #define DLOG(x) PSM_LogDebugInfo x
19165| #else

```

```

19165| #define STATIC static
19166| #define DLOG(x)
19167| #endif
19168|
19169| typedef NTSTATUS (NTAPI *tNtQueryVolumeInformation)(
19170|         IN HANDLE FileHandle,
19171|         OUT PIO_STATUS_BLOCK IoStatusBlock,
19172|         OUT PVOID FsInformation,
19173|         IN ULONG Length,
19174|         IN FS_INFORMATION_CLASS
19175|         | FsInformationClass);
19176|
19176| // specific to Win2k only
19177| typedef BOOL (WINAPI
19178|         | *tGetVolumeNameForVolumeMountPoint)(
19179|         LPWSTR lpszVolumeMountPoint,
19180|         LPWSTR lpszVolumeName,
19181|         DWORD cchBufferLength);
19182|
19182| typedef BOOL (WINAPI *tGetVolumePathName)(
19183|         LPWSTR lpszFileName,
19184|         LPWSTR lpszVolumePathName,
19185|         DWORD cchBufferLength);
19186|
19186| typedef HANDLE (WINAPI *tFindFirstVolume)(
19187|         LPWSTR lpszVolumeName,
19188|         DWORD cchBufferLength
19189|         );
19190|
19191| typedef BOOL (WINAPI *tFindNextVolume)(
19192|         HANDLE hFindVolume,
19193|         LPWSTR lpszVolumeName,
19194|         DWORD cchBufferLength
19195|         );
19196|
19197| typedef BOOL (WINAPI *tFindVolumeClose)(
19198|         HANDLE hFindVolume
19199|         );
19200|
19201|
19202|
19203| typedef PWCHAR *pVolMap;
19204|
19205| typedef struct _MapDrive {
19206|     WCHAR     NTDeviceName[256];
19207|     WCHAR     DriveLetterName[10];
19208|     WCHAR     ShareName[256];
19209|     WCHAR     VolumeName[256];
19210|     WCHAR     OriginalUserName[256];
19211|     WCHAR     OriginalNTName[256];
19212|     WCHAR     Company[80];

```

```

19213|  WCHAR    Product[80];
19214|  WCHAR    Version[80];
19215|  WCHAR    Code[20];
19216|  WCHAR    Key[20];
19217|  ULONG    SnapShotOffset;    // SnapShot
    | associated with this virtual drive
19218| } tMapDrive;
19219|
19220| typedef struct _ThreadStorage {
19221|     ULONG RegisterCalled;
19222|     ULONG Persistent;
19223|     ULONG NumOpens;
19224|     HANDLE PSMAN_EVENT;
19225|     HANDLE PSMAN_HANDLE;
19226|     WCHAR TempFileName[255];
19227|     BOOLEAN UsedTempFile;
19228|     HANDLE AbortEvent;
19229|     WCHAR Company[80];
19230|     WCHAR Product[80];
19231|     WCHAR Version[80];
19232|     WCHAR Code[20];
19233|     WCHAR Key[20];
19234| #ifdef _DEBUG
19235|     HANDLE DebugLogFile;
19236| #endif
19237|     ULONG NumSnapShots;
19238|     ULONG SnapShotsOffset[MAX_NUMBER_OF_SNAPSHOTS];
19239| } tThreadStorage, *pThreadStorage;
19240|
19241| typedef struct sProductEntry {
19242|     struct sProductEntry *Next;
19243|     WCHAR Company[40];
19244|     WCHAR Product[80];
19245|     WCHAR Version[40];
19246|     ULONG Threads;
19247|     ULONG Processes;
19248| } tProductEntry, *pProductEntry;
19249|
19250| typedef struct sSharedMemory {
19251|     ULONG    Size;
19252|     ULONG    NumberOfMappedVolumes;
19253|     tMapDrive MappedVolumes[MAX_VOLUMES_SUPPORTED];
19254|     ULONG    NumberOfSnapShots;
19255|     CHAR
    | SnapShotsInUse[(7+MAX_NUMBER_OF_SNAPSHOTS)/8];
19256|     ULONG
    | SnapShotsOffset[MAX_NUMBER_OF_SNAPSHOTS];
19257|     tSnapShot
    | ActualSnapShotSpace[MAX_NUMBER_OF_SNAPSHOTS];
19258| } tSharedMemory;

```



```

19259|
19260| #define MakePointer(Offset)
    | ((pSnapshot)(((char*)(SharedMemory->ActualSnapshotSpace)
    | )+(Offset)))
19261| #define MakeOffset(Pointer)
    | (ULONG)(((char*)(Pointer))-((char*)SharedMemory->ActualS
    | napShotSpace))
19262|
19263| //-----
    | -----
19264| // Global variables common to all process
19265| STATIC HANDLE MapFileHandle=NULL;
19266| STATIC tSharedMemory *SharedMemory=NULL;
19267|
19268| //-----
    | -----
19269| // global only to this process.
19270| #ifdef _DEBUG
19271| DWORD DebugMode = 0;
19272| #endif
19273| STATIC DWORD TlsIndex = 0xffffffff;
19274| STATIC HANDLE SharedMemoryMutex=INVALID_HANDLE_VALUE;
19275| STATIC HANDLE OpenCloseMutex=INVALID_HANDLE_VALUE;
19276| STATIC ULONG RandomNumberSeed = 1L;
19277| STATIC ULONG CacheFileLocationKeyExists=FALSE;
19278| STATIC WCHAR CacheFileLocation[256]={0};
19279| STATIC WCHAR PreOpenFile[256]={0};
19280| STATIC WCHAR PostOpenFile[256]={0};
19281| STATIC WCHAR PostCloseFile[256]={0};
19282| STATIC WCHAR ChildUser[256]={0};
19283| STATIC WCHAR ChildPassword[256]={0};
19284| STATIC WCHAR SnapShotLocation[256]={0};
19285| STATIC WCHAR SnapShotPattern[256]={0};
19286| STATIC ULONG NtBuildNumber=1381;
19287| STATIC tGetVolumePathName pGetVolumePathName=NULL;
19288| STATIC tGetVolumeNameForVolumeMountPoint
    | pGetVolumeNameForVolumeMountPoint=NULL;
19289| STATIC tNtQueryVolumeInformation
    | pNtQueryVolumeInformation=NULL;
19290| STATIC tFindFirstVolume pFindFirstVolume=NULL;
19291| STATIC tFindNextVolume pFindNextVolume=NULL;
19292| STATIC tFindVolumeClose pFindVolumeClose=NULL;
19293| ULONG GetWin32NameForNtDeviceName( WCHAR *NTName, WCHAR
    | *Win32Name );
19294| STATIC BOOL Login_EnablePrivilege ( LPTSTR Privilege );
19295| extern BOOLEAN HasEvalExpired( pPSM_VersionInfo2
    | Version );
19296|
19297| //-----
    | -----

```

```

19298| // prototypes
19299|
19300| int CreateJunction( PWCHAR LinkDirectory, PWCHAR
    | LinkTarget, PLARGE_INTEGER Time );
19301| int CreateJunction2( PWCHAR LinkDirectory, PWCHAR
    | LinkTarget, PLARGE_INTEGER Time );
19302| int DeleteJunction( PWCHAR Junction );
19303| STATIC pthreadStorage GetThreadStorage();
19304| STATIC pSnapshot AllocSnapshot();
19305| STATIC ULONG IsValidSnapshot( pSnapshot Snapshot );
19306| STATIC void FreeSnapshot( pSnapshot Snapshot );
19307| STATIC ULONG CheckForZeroTerminatorA( char *Str, ULONG
    | Len );
19308| STATIC ULONG CheckForZeroTerminatorW( WCHAR *Str, ULONG
    | Len );
19309| STATIC ULONG PSMI_GetSnapshotInfoFromVolume( WCHAR
    | *VolumeName, tSnapshotInfoW *SnapshotInfo );
19310| STATIC ULONG PSMI_IsAnPSMVolume( WCHAR *VolumeName );
19311| STATIC ULONG PSMI_CanBePSMed( WCHAR *VolumeName, ULONG
    | *VolumeType );
19312| ULONG GetDriveLetterForNtDeviceName( WCHAR *NTName,
    | WCHAR *Win32Name );
19313| ULONG GetVolumeGuidForNtDeviceName( WCHAR *NTName,
    | WCHAR *VolumeGuid);
19314| STATIC int SetTimeForFile( PWCHAR File, PLARGE_INTEGER
    | Time );
19315|
19316| STATIC ULONG PSMI_OpenManager( HANDLE *hDevice );
19317| STATIC ULONG PSMI_OpenEx(
19318|     HANDLE hDevice,
19319|     IN pOpenTransactionIn3W In,
19320|     IN ULONG NumVolumes,
19321|     IN PVOID InVolumeMap,
19322|     IN ULONG *VolumeMapFlags,
19323|     IN ULONG OutVolumeMapByteSize,
19324|     INOUT PVOID OutVolumeMap,
19325|     OUT pOpenTransactionOutW Out,
19326|     INOUT LPOVERLAPPED Overlapped,
19327|     IN LPOVERLAPPED_COMPLETION_ROUTINE
    | CompletionRoutine,
19328|     IN ULONG InternalFlags,
19329|     OUT pSnapshot *Snapshot
19330| );
19331|
19332|
19333| PSMSTATUS PSMAPI Psmi_GetKernelSnapshotVolumesW (
19334|     PVOID KernelSnapshotPointer,
19335|     WCHAR *Buffer,
19336|     ULONG BufferSize );
19337| STATIC ULONG PSMI_GetVolumeStats( HANDLE hDevice,

```

```

    | HANDLE hEvent, WCHAR *VolumeName, tPSM_GetStatsRecord
    | *Stats );
19338| STATIC ULONG GetNTDeviceName( WCHAR *AName, WCHAR
    | *NTName, ULONG BufferLength, ULONG Fast );
19339| STATIC ULONG FreeOneVolumeForSnapShot ( pSnapShot
    | SnapShot, WCHAR *NTName );
19340| STATIC ULONG FreeVolumesForSnapShot ( pSnapShot
    | SnapShot );
19341| STATIC ULONG Random( void );
19342| STATIC ULONG CheckKey( ULONG SerialNumber, ULONG
    | CheckCode );
19343| STATIC ULONG PSMI_Close_Internal( HANDLE hDevice,
    | HANDLE hEvent, PVOID KernelSnapShot );
19344| STATIC ULONG PSMI_Close( HANDLE hDevice, HANDLE hEvent,
    | tSnapShot *SnapShot );
19345| STATIC ULONG PSMI_CloseManager( HANDLE hDevice );
19346| STATIC ULONG PSMI_OpenExclusive ( HANDLE hDevice,
    | HANDLE hEvent, pSnapShot SnapShot );
19347| STATIC ULONG PSMI_CloseExclusive ( HANDLE hDevice,
    | HANDLE hEvent, pSnapShot SnapShot );
19348| STATIC ULONG PSMI_GetProgress ( HANDLE hDevice, HANDLE
    | hEvent, pPSM_GetProgressOut Progress, pSnapShot
    | SnapShot);
19349| STATIC ULONG PSMI_GetVersion ( HANDLE hDevice, HANDLE
    | hEvent, ULONG VerSize, pPSM_VersionInfo Version );
19350| STATIC ULONG PSMI_FreeFiles( HANDLE hDevice, HANDLE
    | hEvent, pSnapShot SnapShot, ULONG NumberOfFiles, WCHAR
    | *Files[] );
19351| STATIC ULONG PSMI_FreeRanges ( HANDLE hDevice, HANDLE
    | hEvent, tPSM_FreeRanges *Ranges);
19352| STATIC ULONG PSMI_FreeVolume ( HANDLE hDevice, HANDLE
    | hEvent, tPSM_FreeVolume *Volume);
19353| STATIC ULONG RemoveAllVDsForThread();
19354| //STATIC ULONG AddDrivesToSystem( tAPCContext
    | *ApcContext );
19355| STATIC ULONG PSMI_GetError ( HANDLE hDevice, HANDLE
    | hEvent, pPSM_GetErrorOut Error, pSnapShot SnapShot);
19356| STATIC int ExecuteFile ( WCHAR *FileName, WCHAR
    | *CommandLine, WCHAR *ChildUser, WCHAR *ChildPassword );
19357| DWORD ExceptionFilter( EXCEPTION_POINTERS *ep );
19358| PSMSTATUS PSMAPI PSMI_LogEvent( ULONG EventId, ULONG
    | Status, ULONG NumStrings, WCHAR *Strings[] );
19359| ULONG GetOurPath( WCHAR *OurPath );
19360| ULONG GetUniqueIdForVolume (HANDLE VolHandle, WCHAR
    | *UniqueId,ULONG UniqueIdLength);
19361| ULONG MakeVolumeList (
19362|     ULONG NumVolumes,
19363|     pVolMap VolumeMap,
19364|     ULONG BufferLength,
19365|     pOpenTransactionInternal Internal,

```

```

19366|    ULONG InternalBufferChars );
19367|
19368| #define WINDOWS_NT_UNKNOWN        0x00
19369| #define WINDOWS_NT_WORKSTATION    0x01
19370| #define WINDOWS_NT_SERVER         0x02
19371| #define WINDOWS_NT_DOMAIN         0x04
19372| #define WINDOWS_NT_ENTERPRISE     0x08
19373| #define WINDOWS_NT_TERMINAL_SERVER 0x10
19374|
19375| STATIC int GetNTSystemType( );
19376|
19377| STATIC WCHAR _COPYRIGHT_STRINGS_STRING[] = L"Copyright
    | 1999-2001 Columbia Data Products, Inc. All Rights
    | Reserved.\n"
19378|                L"The
    | following 3.2k cannot be used without permission.\n";
19379| STATIC WCHAR *_C_STRINGS[] = {
19380|
19381| // Redacted for confidentiality. Redaction does
19382| // not teach any claimed material. --LPW
19383|
19384| };
19385|
19386| //-----
    | -----
19387| // Public APIs
19388|
19389| //-----
    | -----
19390|
19391| DLLEXPORT BOOLEAN PSMAPI
    | Psm_IsPersistentSnapShotPointer ( IN pSnapShot snapshot
    | )
19392| {
19393|     BOOLEAN isPersistent = FALSE;
19394|     if ( (ULONG)snapshot & 0x80000000 ) {
19395|         isPersistent = TRUE;
19396|     }
19397|     return isPersistent;
19398| }
19399|
19400| void RemoveLineFeeds(WCHAR *String)
19401| {
19402|     ULONG i;
19403|     for(i=0;i<wcslen(String);i) {
19404|         if((String[i]==L'\n') || (String[i]==L'\r')) {
19405|             memmove(&String[i],&String[i+1],wcslen(&String[i+1]));
19406|             String[wcslen(String)-1]=0;
19407|         } else {

```

```

19408|         i++;
19409|     }
19410| }
19411| }
19412|
19413|
19414| ULONG GetStringFromId( WCHAR *PathToUs, ULONG Id, WCHAR
    | *String, ULONG Count )
19415| {
19416| #define RESOURCE_WAY 0
19417| #if RESOURCE_WAY
19418|     ULONG Err;
19419|
19420|     HMODULE Instance=GetModuleHandleW(PathToUs);
19421|     if(Instance) {
19422|         if(LoadStringW(Instance,Id,String,Count)==0) {
19423|             Err = GetLastError();
19424|         }
19425|     } else {
19426|         Err = GetLastError();
19427|         DLOG((TEXT("Error %08x getting module
    | handle\n"),Err));
19428|         Instance=LoadLibraryW(PathToUs);
19429|         if(Instance) {
19430|             Err=0;
19431|
    | if(LoadStringW(Instance,Id,String,Count)==0) {
19432|                 Err = GetLastError();
19433|             }
19434|             FreeLibrary(Instance);
19435|         } else {
19436|             Err = GetLastError();
19437|             DLOG((TEXT("Error %08x getting library
    | module handle\n"),Err));
19438|         }
19439|     }
19440|     return Err;
19441| #else
19442|     ULONG Err;
19443|     HMODULE Instance=GetModuleHandleW(PathToUs);
19444|     if(Instance) {
19445|         if(!FormatMessageW(
19446|             FORMAT_MESSAGE_FROM_HMODULE |
    | FORMAT_MESSAGE_IGNORE_INSERTS, // flags
19447|             Instance, // message source
19448|             Id, // message id
19449|             0, // lang id, MAKELANGID(LANG_NEUTRAL,
    | SUBLANG_DEFAULT)
19450|             String, // buffer
19451|             Count, // size of buffer

```

```

19452|         NULL // array of message inserts
19453|     )) {
19454|         Err = GetLastError();
19455|     } else {
19456|         Err=0;
19457|         RemoveLineFeeds(String);
19458|     }
19459| } else {
19460|     Err = GetLastError();
19461|     DLOG((TEXT("Error %08x getting module
    | handle\n"),Err));
19462|     Instance=LoadLibraryW(PathToUs);
19463|     if(Instance) {
19464|         if(!FormatMessageW(
19465|             FORMAT_MESSAGE_FROM_HMODULE |
    | FORMAT_MESSAGE_IGNORE_INSERTS, // flags
19466|             Instance, // message source
19467|             Id, // message id
19468|             0, // lang id, MAKELANGID(LANG_NEUTRAL,
    | SUBLANG_DEFAULT)
19469|             String, // buffer
19470|             Count, // size of buffer
19471|             NULL // array of message inserts
19472|             )) {
19473|                 Err = GetLastError();
19474|             } else {
19475|                 Err=0;
19476|                 RemoveLineFeeds(String);
19477|             }
19478|             FreeLibrary(Instance);
19479|         } else {
19480|             Err = GetLastError();
19481|             DLOG((TEXT("Error %08x getting library
    | module handle\n"),Err));
19482|         }
19483|     }
19484|     return Err;
19485| #endif
19486| }
19487|
19488| DLLEXPORT PSMSTATUS PSMAPI Psm_GetStatusW ( OUT
    | tPSM_GetStatusOutW *Status, IN ULONG Count, IN ULONG
    | Type)
19489| {
19490|     ULONG DataSize;
19491|     HKEY Key;
19492|     ULONG Err;
19493|     TCHAR RegStr[255];
19494|     WCHAR wTmpStr[255]={0};
19495|     WCHAR wTmpStr2[255]={0};

```

```

19496|  WCHAR Path[1024]={0};
19497|
19498|  // Clear Status Structure
19499|  wcsncpy(Status->StatusMessage,L"");
19500|  Status->StatusCode=PSM_ERROR_UNSUCCESSFUL;
19501|
19502|  // First Get Location of psmlapi Path
19503|
19504|  | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
19505|  | es\\PSMan%d\\persistent"), NtBuildNumber > 1381 ? 5 :
19506|  | 4);
19507|  // open the registry
19508|  Err = RegOpenKeyEx(
19509|      HKEY_LOCAL_MACHINE, // handle of open key
19510|      RegStr, // address of name of subkey to open
19511|      0, // reserved
19512|      KEY_READ, // security access mask
19513|      &Key// address of handle of open key
19514|  );
19515|
19516|  if(Err==0) {
19517|      DataSize = 256;
19518|
19519|      Err =
19520|      | RegQueryValueExW(Key,L"SnapShotInitiator",
19521|      | NULL,NULL,(unsigned char
19522|      | *)wTmpStr,&DataSize );
19523|
19524|      | ExpandEnvironmentStringsW(wTmpStr,wTmpStr2,255);
19525|      // Remove Initiator Executable
19526|      wTmpStr2[wcslen(wTmpStr2)-6]='\0';
19527|      // Append psmlapi.dll
19528|      wcscat(wTmpStr2,L"psmlapi.dll");
19529|
19530|      // close the registry
19531|      RegCloseKey(Key);
19532|  }
19533|
19534|  switch(Type){
19535|      case PSM_GLOBAL_STATUS:
19536|      case PSM_REVERT_STATUS:
19537|
19538|      | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
19539|      | es\\PSMan%d\\persistent"), NtBuildNumber > 1381 ? 5 :
19540|      | 4);
19541|
19542|      break;
19543|      case PSM_ENGINE_STATUS:
19544|
19545|      | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic

```

```

    | es\\PSMan%d\\Backup"), NtBuildNumber > 1381 ? 5 : 4);
19536|         break;
19537|         case PSM_PATH_LOCATION1_STATUS:
19538|         case PSM_RESULT_LOCATION1_STATUS:
19539|
    | _sprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
    | es\\PSMan%d\\Backup\\Backup1"), NtBuildNumber > 1381 ?
    | 5 : 4);
19540|         break;
19541|         case PSM_PATH_LOCATION2_STATUS:
19542|         case PSM_RESULT_LOCATION2_STATUS:
19543|
    | _sprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
    | es\\PSMan%d\\Backup\\Backup2"), NtBuildNumber > 1381 ?
    | 5 : 4);
19544|         break;
19545|         case PSM_PATH_LOCATION3_STATUS:
19546|         case PSM_RESULT_LOCATION3_STATUS:
19547|
    | _sprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
    | es\\PSMan%d\\Backup\\Backup3"), NtBuildNumber > 1381 ?
    | 5 : 4);
19548|         break;
19549|         case PSM_PATH_DISKETTE_STATUS:
19550|         case PSM_RESULT_DISKETTE_STATUS:
19551|
    | _sprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
    | es\\PSMan%d\\Backup\\Diskette"), NtBuildNumber > 1381 ?
    | 5 : 4);
19552|         break;
19553|     }
19554|     // open the registry
19555|     Err = RegOpenKeyEx(
19556|         HKEY_LOCAL_MACHINE, // handle of open key
19557|         RegStr, // address of name of subkey to open
19558|         0, // reserved
19559|         KEY_READ, // security access mask
19560|         &Key// address of handle of open key
19561|     );
19562|
19563|     if(Err==0) {
19564|         ULONG Id;
19565|
19566|         DataSize = 4;
19567|         switch(Type) {
19568|             case PSM_GLOBAL_STATUS:
19569|                 Err =
    | RegQueryValueExW(Key,L"GlobalStatus",NULL,NULL,(char*)&I
    | d,&DataSize );
19570|         break;

```



```

19571|         case PSM_REVERT_STATUS:
19572|             Err =
19573|             | RegQueryValueExW(Key,L"RevertStatus",NULL,NULL,(char*)&Id
19574|             | d,&DataSize );
19575|             break;
19576|         case PSM_ENGINE_STATUS:
19577|             Err =
19578|             | RegQueryValueExW(Key,L"EngineStatus",NULL,NULL,(char*)&Id
19579|             | d,&DataSize );
19580|             break;
19581|         case PSM_PATH_LOCATION1_STATUS:
19582|         case PSM_PATH_LOCATION2_STATUS:
19583|         case PSM_PATH_LOCATION3_STATUS:
19584|         case PSM_PATH_DISKETTE_STATUS:
19585|             Err =
19586|             | RegQueryValueExW(Key,L"PathStatus",NULL,NULL,(char*)&Id,
19587|             | &DataSize );
19588|             break;
19589|         case PSM_RESULT_LOCATION1_STATUS:
19590|         case PSM_RESULT_LOCATION2_STATUS:
19591|         case PSM_RESULT_LOCATION3_STATUS:
19592|             Err =
19593|             | RegQueryValueExW(Key,L"LastBackupResult",NULL,NULL,(char
19594|             | *)&Id,&DataSize );
19595|             break;
19596|         case PSM_RESULT_DISKETTE_STATUS:
19597|             Err =
19598|             | RegQueryValueExW(Key,L"LastUpdateResult",NULL,NULL,(char
19599|             | *)&Id,&DataSize );
19600|             break;
19601|         }
19602|         if(!Err) {
19603|             ULONG Err = GetOurPath(Path);
19604|             wscpy(Path,wTmpStr2);
19605|             DLOG((TEXT("Found us at '%S'\n"),Path));
19606|             if(!Err) {
19607|                 Status->StatusCode=Id;
19608|                 Err = GetStringFromId( Path, Id,
19609|                 | Status->StatusMessage, Count );
19610|             }
19611|         }
19612|         // close the registry
19613|         RegCloseKey(Key);
19614|     }
19615| }
19616| return Err;
19617| }
19618|

```

```

19610| DLLEXPORT PSMSTATUS PSMAPI Psm_GetStatusA ( OUT
    | tPSM_GetStatusOutA *Status, IN ULONG Count, IN ULONG
    | Type)
19611| {
19612|     tPSM_GetStatusOutW StatusW;
19613|     ULONG Err;
19614|     Err = Psm_GetStatusW(&StatusW,Count,Type);
19615|     if(!Err) {
19616|         | CharToOemW(StatusW.StatusMessage,Status->StatusMessage);
19617|         Status->StatusCode=StatusW.StatusCode;
19618|     }
19619|     return Err;
19620| }
19621|
19622| DLLEXPORT PSMSTATUS PSMAPI Psm_SetStatus ( IN ULONG
    | StatusCode, IN ULONG Type)
19623| {
19624|     HKEY Key;
19625|     ULONG Err;
19626|     TCHAR RegStr[255];
19627|
19628|     switch(Type){
19629|         case PSM_GLOBAL_STATUS:
19630|         case PSM_REVERT_STATUS:
19631|
            | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
            | es\\PSMan%d\\persistent"), NtBuildNumber > 1381 ? 5 :
            | 4);
19632|
19633|         break;
19634|         case PSM_ENGINE_STATUS:
19635|
            | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
            | es\\PSMan%d\\Backup"), NtBuildNumber > 1381 ? 5 : 4);
19636|         break;
19637|         case PSM_PATH_LOCATION1_STATUS:
19638|         case PSM_RESULT_LOCATION1_STATUS:
19639|
            | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
            | es\\PSMan%d\\Backup\\Backup1"), NtBuildNumber > 1381 ?
            | 5 : 4);
19640|         break;
19641|         case PSM_PATH_LOCATION2_STATUS:
19642|         case PSM_RESULT_LOCATION2_STATUS:
19643|
            | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
            | es\\PSMan%d\\Backup\\Backup2"), NtBuildNumber > 1381 ?
            | 5 : 4);
19644|         break;

```

```

19645|     case PSM_PATH_LOCATION3_STATUS:
19646|     case PSM_RESULT_LOCATION3_STATUS:
19647|         | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
         | es\\PSMan%d\\Backup\\Backup3"), NtBuildNumber > 1381 ?
         | 5 : 4);
19648|         break;
19649|     case PSM_PATH_DISKETTE_STATUS:
19650|     case PSM_RESULT_DISKETTE_STATUS:
19651|         | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
         | es\\PSMan%d\\Backup\\Diskette"), NtBuildNumber > 1381 ?
         | 5 : 4);
19652|         break;
19653|     }
19654|     // open the registry
19655|     Err = RegOpenKeyEx(
19656|         HKEY_LOCAL_MACHINE, // handle of open key
19657|         RegStr, // address of name of subkey to open
19658|         0, // reserved
19659|         KEY_READ|KEY_WRITE, // security access mask
19660|         &Key// address of handle of open key
19661|     );
19662|
19663|     if(Err==0) {
19664|
19665|         switch(Type) {
19666|             case PSM_GLOBAL_STATUS:
19667|                 Err =
         | RegSetValueEx(Key,TEXT("GlobalStatus"),0,REG_DWORD,(BYTE
         | *)&StatusCode,sizeof(DWORD));
19668|                 break;
19669|             case PSM_REVERT_STATUS:
19670|                 Err =
         | RegSetValueEx(Key,TEXT("RevertStatus"),0,REG_DWORD,(BYTE
         | *)&StatusCode,sizeof(DWORD));
19671|                 break;
19672|             case PSM_ENGINE_STATUS:
19673|                 Err =
         | RegSetValueEx(Key,TEXT("EngineStatus"),0,REG_DWORD,(BYTE
         | *)&StatusCode,sizeof(DWORD));
19674|                 break;
19675|             case PSM_PATH_LOCATION1_STATUS:
19676|             case PSM_PATH_LOCATION2_STATUS:
19677|             case PSM_PATH_LOCATION3_STATUS:
19678|             case PSM_PATH_DISKETTE_STATUS:
19679|                 Err =
         | RegSetValueEx(Key,TEXT("PathStatus"),0,REG_DWORD,(BYTE*)
         | &StatusCode,sizeof(DWORD));
19680|                 break;

```

```

19681|         case PSM_RESULT_LOCATION1_STATUS:
19682|         case PSM_RESULT_LOCATION2_STATUS:
19683|         case PSM_RESULT_LOCATION3_STATUS:
19684|             Err =
        | RegSetValueEx(Key,TEXT("LastBackupResult"),0,REG_DWORD,(
        | BYTE*)&StatusCode,sizeof(DWORD));
19685|             break;
19686|         case PSM_RESULT_DISKETTE_STATUS:
19687|             Err =
        | RegSetValueEx(Key,TEXT("LastUpdateResult"),0,REG_DWORD,(
        | BYTE*)&StatusCode,sizeof(DWORD));
19688|             break;
19689|     }
19690|     // close the registry
19691|     RegCloseKey(Key);
19692| }
19693|
19694| return Err;
19695| }
19696|
19697|
19698| DLLEXPORT PSMSTATUS PSMAPI Psm_GetVolumeUIDW ( IN WCHAR
        | *wVolGUID, OUT WCHAR *wVolUID)
19699| {
19700|     ULONG Err=0;
19701|     UNICODE_STRING Uni;
19702|     HANDLE hVolume;
19703|     OBJECT_ATTRIBUTES ObjectAttributes={0};
19704|     IO_STATUS_BLOCK IoStatus={0};
19705|
19706|     RtlInitUnicodeString( &Uni, wVolGUID);
19707|
19708|     InitializeObjectAttributes ( &ObjectAttributes,
19709|                                 &Uni,
19710|                                 OBJ_CASE_INSENSITIVE,
19711|                                 NULL,
19712|                                 NULL );
19713|     Err = NtCreateFile( &hVolume,
19714|                        FILE_GENERIC_READ,
        | // desired access
19715|                        &ObjectAttributes,
        | // object attributes
19716|                        &IoStatus,
19717|                        NULL,
        | // alloc size
19718|                        FILE_ATTRIBUTE_NORMAL,
        | // file attributes
19719|                        FILE_SHARE_WRITE |
        | FILE_SHARE_READ, // share access
19720|                        FILE_OPEN,

```

```

    | // create disposition
19721|             FILE_SYNCHRONOUS_IO_NONALERT,
    | // create options
19722|             NULL,
    | // eabuffer
19723|             0 );
    | // ealength
19724|
19725|     if(!Err) {
19726|         Err =
            | GetUniqueIdForVolume(hVolume,wVolUID,256);
19727|
19728|         if(!Err) {
19729|             DLOG((TEXT("Volume '%S' unique id =
                | '%S'\n"),wVolGUID,wVolUID));
19730|         }
19731|         else {
19732|             DLOG((TEXT("Error %08x getting unique
                | id\n"),Err));
19733|         }
19734|         NtClose(hVolume);
19735|     }
19736|
19737|     return Err;
19738| }
19739|
19740| DLLEXPORT PSMSTATUS PSMAPI Psm_GetVolumeUIDA ( IN char
    | *VolumeGUID, OUT char *VolumeUniqueId)
19741| {
19742|     ULONG Err=0;
19743|
19744|     return Err;
19745| }
19746|
19747| //-----
    | -----
19748| DLLEXPORT PSMSTATUS PSMAPI Psm_RegisterW(
19749|             WCHAR
    | *Company,
19750|             WCHAR
    | *Product,
19751|             WCHAR
    | *Version OPTIONAL,
19752|             WCHAR
    | *Code,
19753|             WCHAR *Key
    | OPTIONAL
19754|             )
19755| {
19756|     ULONG Err=PSM_ERROR_LICENSE_INVALID;

```

```

19757|   pThreadStorage ThreadStorage = GetThreadStorage();
19758|
19759|   __try {
19760|       ULONG Size = sizeof(_C_STRINGS) /
        | sizeof(_C_STRINGS[0]);
19761|       ULONG i;
19762|       DLOG((TEXT("Psm_Register called code = '%ws',
        | key = '%ws'\n"),Code,Key));
19763|
19764|       wcsncpy(ThreadStorage->Company,Company,79);
19765|       wcsncpy(ThreadStorage->Product,Product,79);
19766|       wcsncpy(ThreadStorage->Code,Code,19);
19767|       if(Version)
19768|           wcsncpy(ThreadStorage->Version,Version,79);
19769|       if(Key)
19770|           wcsncpy(ThreadStorage->Key,Key,19);
19771|
19772|
19773|       for(i=0;i<Size;i++) {
19774|           if(wcsncmp(_C_STRINGS[i],Code)==0) {
19775|               switch (_C_STRINGS[i][0]) {
19776|                   case L'0' :
19777|                       ThreadStorage->RegisterCalled =
        | TRUE;
19778|                       Err = 0;
19779|                       break;
19780|                   case L'1' :
19781|                       if(Key) {
19782|                           ULONG SerialNumber;
19783|                           ULONG CheckCode;
19784|
19785|                           // key = PPSSSSSS-CCCCCCCC
        | where
19786|                           // PP = Oem/Product
19787|                           // SS = Serial number for
        | PP
19788|                           // CC = Check Code
19789|                           if( (Code[6]==Key[0]) &&
19790|                               (Code[7]==Key[1])) {
19791|                               // code matches oem and
        | product
19792|                               i =
        | swscanf(Key,L"%x-%x",&SerialNumber,&CheckCode);
19793|                               if((i==2) &&
        | (SerialNumber!=0) && (CheckCode!=0)) {
19794|                               | if(CheckKey(SerialNumber,CheckCode)) {
19795|                               | ThreadStorage->RegisterCalled = TRUE;
19796|                               Err = 0;

```

```

19797|             break;
19798|         }
19799|     }
19800| }
19801| }
19802|     break;
19803|     default:
19804|         break;
19805|     }
19806|     break;
19807| }
19808| }
19809| } __except
| (ExceptionFilter(GetExceptionInformation())) {
19810|     Err = GetExceptionCode();
19811|     DLOG((TEXT("Exception %08x
| Registering\n"),Err));
19812| }
19813|
19814| return Err;
19815| }
19816|
19817| DLLEXPORT PSMSTATUS PSMAPI Psm_InstallPsm( BOOLEAN
| *RebootNeeded )
19818| {
19819|     if(NtBuildNumber > 1381) {
19820|         // install Win2000 way
19821|         return SetUpForWin2k( RebootNeeded );
19822|     } else {
19823|         // install NT 3.51/4.x way
19824|         return SetUpForWinNT( RebootNeeded );
19825|     }
19826| }
19827|
19828| DLLEXPORT PSMSTATUS PSMAPI Psm_UnInstallPsm( BOOLEAN
| *RebootNeeded )
19829| {
19830|     if(NtBuildNumber > 1381) {
19831|         // uninstall Win2000 way
19832|         return UnSetUpForWin2k( RebootNeeded );
19833|     } else {
19834|         // uninstall NT 3.51/4.x way
19835|         return UnSetUpForWinNT( RebootNeeded );
19836|     }
19837| }
19838|
19839| DLLEXPORT PSMSTATUS PSMAPI Psm_Config ( tPSM_Config
| *Config, CHAR Set )
19840| {
19841|     ULONG DataSize;

```

```

19842| HKEY Key;
19843| ULONG Err;
19844| TCHAR RegStr[255];
19845|
19846|
19847| | _sprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
19848| | es\\PSMan%d"), NtBuildNumber > 1381 ? 5 : 4);
19849|
19850| | if(!Config) ||
19851| | (Config->Size!=sizeof(tPSM_Config))) {
19852| |     return ERROR_INVALID_PARAMETER;
19853| | }
19854|
19855| // open the registry
19856| Err = RegOpenKeyEx(
19857| HKEY_LOCAL_MACHINE, // handle of open key
19858| RegStr, // address of name of subkey to open
19859| 0, // reserved
19860| Set ? KEY_WRITE : KEY_READ, // security access
19861| | mask
19862| &Key// address of handle of open key
19863| );
19864|
19865| if(Err==0) {
19866|     if(Set) {
19867|         Err = RegSetValueEx(Key,TEXT("NumThreads")
19868|         | ,0,REG_DWORD,(BYTE*)&Config->StartingThreadCount,sizeof(
19869|         | DWORD));
19870|         Err = RegSetValueEx(Key,TEXT("MaxThreads")
19871|         | ,0,REG_DWORD,(BYTE*)&Config->MaximumThreadCount,sizeof(D
19872|         | WORD));
19873|         Err =
19874|         | RegSetValueEx(Key,TEXT("HungSystemTimeOut")
19875|         | ,0,REG_DWORD,(BYTE*)&Config->DeadlockTimeout,sizeof(DWOR
19876|         | D));
19877|         Err =
19878|         | RegSetValueEx(Key,TEXT("NewThreadStartDelay")
19879|         | ,0,REG_DWORD,(BYTE*)&Config->NewThreadDelay,sizeof(DWORD
19880|         | ));
19881|     } else {
19882|         DataSize = 4;
19883|         Err = RegQueryValueExW(Key,L"NumThreads"
19884|         | ,NULL,NULL,(char*)&Config->StartingThreadCount,&DataSize
19885|         | );
19886|         Err = RegQueryValueExW(Key,L"MaxThreads"
19887|         | ,NULL,NULL,(char*)&Config->MaximumThreadCount,&DataSize)
19888|         | ;
19889|         Err =
19890|         | RegQueryValueExW(Key,L"HungSystemTimeOut"
19891|         | ,NULL,NULL,(char*)&Config->DeadlockTimeout,&DataSize);

```



```

19872|         Err =
    | RegQueryValueExW(Key,L"NewThreadStartDelay"
    | ,NULL,NULL,(char*)&Config->NewThreadDelay,&DataSize);
19873|     }
19874|     // close the registry
19875|     RegCloseKey(Key);
19876| }
19877| return Err;
19878| }
19879|
19880|
19881| DLLEXPORT PSMSTATUS PSMAPI Psm_RegisterA(
19882|         CHAR
    | *Company,
19883|         CHAR
    | *Product,
19884|         CHAR
    | *Version OPTIONAL,
19885|         CHAR *Code,
19886|         CHAR *Key
    | OPTIONAL
19887|         )
19888| {
19889|     WCHAR *CompanyW=NULL;
19890|     WCHAR *ProductW=NULL;
19891|     WCHAR *VersionW=NULL;
19892|     WCHAR *CodeW=NULL;
19893|     WCHAR *KeyW=NULL;
19894|     ULONG Err;
19895|
19896|     if(Company) {
19897|         CompanyW =
            | LocalAlloc(LPTR,strlen(Company)*2+2);
19898|         OemToCharW(Company,CompanyW);
19899|     }
19900|     if(Product) {
19901|         ProductW= LocalAlloc(LPTR,strlen(Product)*2+2);
19902|         OemToCharW(Product,ProductW);
19903|     }
19904|     if(Version) {
19905|         VersionW =
            | LocalAlloc(LPTR,strlen(Version)*2+2);
19906|         OemToCharW(Version,VersionW);
19907|     }
19908|     if(Code) {
19909|         CodeW = LocalAlloc(LPTR,strlen(Code)*2+2);
19910|         OemToCharW(Code,CodeW);
19911|     }
19912|     if(Key) {
19913|         KeyW = LocalAlloc(LPTR,strlen(Key)*2+2);

```

```

19914|     OemToCharW(Key,KeyW);
19915| }
19916|
19917| Err =
    | Psm_RegisterW(CompanyW,ProductW,VersionW,CodeW,KeyW);
19918|
19919| if(CompanyW) {
19920|     LocalFree(CompanyW);
19921| }
19922| if(ProductW) {
19923|     LocalFree(ProductW);
19924| }
19925| if(VersionW) {
19926|     LocalFree(VersionW);
19927| }
19928| if(CodeW) {
19929|     LocalFree(CodeW);
19930| }
19931| if(KeyW) {
19932|     LocalFree(KeyW);
19933| }
19934| return Err;
19935| }
19936|
19937| DLLEXPORT PSMSTATUS PSMAPI Psm_IsInstalled(
19938|     IN ULONG VerSize,
19939|     OUT pPSM_VersionInfo2 Version )
19940| {
19941|     ULONG Err=ERROR_INVALID_HANDLE;
19942|     ULONG Type;
19943|     pThreadStorage ThreadStorage = GetThreadStorage();
19944|
19945|     __try {
19946|         DLOG((TEXT("Psm_IsInstalled called\n")));
19947|
19948|         if( ((VerSize == sizeof(tPSM_VersionInfo1)) ||
            | (VerSize == sizeof(tPSM_VersionInfo2)))) {
19949|             Version->Size =VerSize;
19950|             Version->Version = 0;
19951|
19952|             if
            | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
            | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {
19953|                 Err =
                    | PSMI_GetVersion(ThreadStorage->PSManHandle,ThreadStorage
                    | ->PSManEvent,VerSize,Version);
19954|                 if(Version->Size >=
                    | sizeof(tPSM_VersionInfo2)) {
19955|                     // get eval information
19956|                     HasEvalExpired(Version);

```

```

19957|         }
19958|     } else {
19959|         TCHAR VersionToDo[20];
19960|
19961|         _sprintf(VersionToDo,TEXT("PSMan%d"),
| NtBuildNumber > 1381 ? 5 : 4);
19962|
19963|         if (Svc_CheckServiceExists( VersionToDo
| )==0) {
19964|             // could not open driver but it is
| installed, must need a reboot
19965|             Err = PSM_ERROR_REBOOT_NEEDED;
19966|         } else {
19967|             // psm is not installed.
19968|             Err = PSM_ERROR_NOT_INSTALLED;
19969|         }
19970|     }
19971|
19972|     // fill in version info about this box even
| if there is an error.
19973|     Type = GetNTSystemType();
19974|
19975|     // we know we are on NT so say WS until we
| can figure out what flavor.
19976|     Version->OSType = PSM_OS_WIN_NT_WS;
19977|
19978|     if(Type & WINDOWS_NT_SERVER)
19979|         Version->OSType = PSM_OS_WIN_NT_SERVER;
19980|     if(Type & WINDOWS_NT_DOMAIN)
19981|         Version->OSType = PSM_OS_WIN_NT_DOMAIN;
19982|     if(Type & WINDOWS_NT_ENTERPRISE)
19983|         Version->OSType = PSM_OS_WIN_NT_ENT;
19984|     if(Type & WINDOWS_NT_TERMINAL_SERVER)
19985|         Version->OSType = PSM_OS_WIN_NT_TS;
19986|
19987|     Version->OSVersion = GetVersion();
19988| } else {
19989|     // wrong size...
19990|     Err = ERROR_INVALID_PARAMETER;
19991| }
19992| } __except
| (ExceptionFilter(GetExceptionInformation())) {
19993|     Err = GetExceptionCode();
19994|     DLOG((TEXT("Exception %08x getting
| version\n"),Err));
19995| }
19996|
19997| return Err;
19998| }
19999|

```

```

20000| DLLEXPORT PSMSTATUS PSMAPI Psm_ListPSMSnapShots( ULONG
      | Iterator )
20001| {
20002| //  return PSMI_ListPSMSnapShots(iterator);
20003|  return 0;
20004| }
20005|
20006| DLLEXPORT PSMSTATUS PSMAPI Psm_GetPersistentSnapShots (
20007|  PVOID Buffer,
20008|  ULONG BufferSize )
20009| {
20010|  ULONG B=FALSE;
20011|  LONG Err=0;
20012|  OVERLAPPED o;
20013|  ULONG returned=0;
20014|  pThreadStorage ThreadStorage = GetThreadStorage();
20015|
20016|  __try {
20017|      if
      | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
20018|          memset(&o,0,sizeof(o));
20019|          o.hEvent = CreateEvent( NULL, FALSE, FALSE,
      | NULL );
20020|
20021|          B = DeviceIoControl(
      | ThreadStorage->PSManHandle,
20022|          IOCTL_GET_PERSISTENT_SNAPSHOTS,
20023|          NULL,
20024|          0,
20025|          Buffer,
20026|          BufferSize,
20027|          &returned,
20028|          &o);
20029|
20030|          if(B) {
20031|              Err = 0;
20032|          } else {
20033|              Err = GetLastError();
20034|              DLOG((TEXT("Error %08x Calling
      | Psm_GetPersistentSnapShots\n"),Err));
20035|          }
20036|          CloseHandle(o.hEvent);
20037|      } else {
20038|          Err = ERROR_INVALID_HANDLE;
20039|      }
20040|  } __except(EXCEPTION_EXECUTE_HANDLER) {
20041|      Err = GetExceptionCode();
20042|      DLOG((TEXT("Exception %08x
      | Psm_GetPersistentSnapShots\n"),Err));
20043|  }

```

```

20044|
20045|     return Err;
20046| }
20047|
20048| DLLEXPORT PSMSTATUS PSMAPI Psm_GetUserName(
20049|     PVOID KernelSnapShotPointer,
20050|     PVOID Buffer,
20051|     ULONG BufferSize )
20052| {
20053|     ULONG B=FALSE;
20054|     LONG Err=0;
20055|     OVERLAPPED o;
20056|     ULONG returned=0;
20057|     pThreadStorage ThreadStorage = GetThreadStorage();
20058|
20059|     __try {
20060|         if
20061|         | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
20062|             memset(&o,0,sizeof(o));
20063|             o.hEvent = CreateEvent( NULL, FALSE, FALSE,
20064|             | NULL );
20065|             B = DeviceIoControl(
20066|             | ThreadStorage->PSManHandle,
20067|             | IOCTL_GET_USER_NAME,
20068|             | &KernelSnapShotPointer,
20069|             | sizeof(PVOID),
20070|             | Buffer,
20071|             | BufferSize,
20072|             | &returned,
20073|             | &o);
20074|             if(B) {
20075|                 Err = 0;
20076|             } else {
20077|                 Err = GetLastError();
20078|                 DLOG((TEXT("Error %08x Calling
20079|                 | Psm_GetUserName\n"),Err));
20080|             }
20081|             CloseHandle(o.hEvent);
20082|         } else {
20083|             Err = ERROR_INVALID_HANDLE;
20084|         }
20085|     } __except(EXCEPTION_EXECUTE_HANDLER) {
20086|         Err = GetExceptionCode();
20087|         DLOG((TEXT("Exception %08x
20088|         | Psm_GetUserName\n"),Err));
20089|     }
20090|     return Err;

```

```

20089| }
20090|
20091| DLLEXPORT PSMSTATUS PSMAPI Psm_SetUserName(
20092|     PVOID KernelSnapShotPointer,
20093|     PVOID Buffer,
20094|     ULONG BufferSize )
20095| {
20096|     ULONG B=FALSE;
20097|     LONG Err=0;
20098|     OVERLAPPED o;
20099|     ULONG returned=0;
20100|     pThreadStorage ThreadStorage = GetThreadStorage();
20101|     tPSM_SetUserNameIn In;
20102|
20103|     __try {
20104|         if
            | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
20105|             memset(&o,0,sizeof(o));
20106|             o.hEvent = CreateEvent( NULL, FALSE, FALSE,
            | NULL );
20107|
20108|             In.KernelSnapShotPointer =
            | KernelSnapShotPointer;
20109|             if(wcslen(Buffer)<256) {
20110|                 wcsncpy(In.Name,Buffer);
20111|
20112|                 B = DeviceIoControl(
            | ThreadStorage->PSManHandle,
20113|                 IOCTL_SET_USER_NAME,
20114|                 &In,
20115|                 sizeof(In),
20116|                 NULL,
20117|                 0,
20118|                 &returned,
20119|                 &o);
20120|
20121|                 if(B) {
20122|                     Err = 0;
20123|                 } else {
20124|                     Err = GetLastError();
20125|                     DLOG((TEXT("Error %08x Calling
            | Psm_SetUserName\n"),Err));
20126|                 }
20127|             } else {
20128|                 Err = ERROR_INVALID_PARAMETER;
20129|             }
20130|             CloseHandle(o.hEvent);
20131|         } else {
20132|             Err = ERROR_INVALID_HANDLE;
20133|         }

```

```

20134| } __except(EXCEPTION_EXECUTE_HANDLER) {
20135|     Err = GetExceptionCode();
20136|     DLOG((TEXT("Exception %08x
    | Psm_SetUserName\n"),Err));
20137| }
20138|
20139| return Err;
20140| }
20141|
20142| DLLEXPORT PSMSTATUS PSMAPI
    | Psm_RevertSnapShotToPristineState(
20143|     PVOID KernelSnapShotPointer )
20144| {
20145|     ULONG B=FALSE;
20146|     LONG Err=0;
20147|     OVERLAPPED o;
20148|     ULONG returned=0;
20149|     pThreadStorage ThreadStorage = GetThreadStorage();
20150|     tPSM_SnapShotPointer In;
20151|
20152|     __try {
20153|         if
            | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
20154|             memset(&o,0,sizeof(o));
20155|             o.hEvent = CreateEvent( NULL, FALSE, FALSE,
            | NULL );
20156|
20157|             In.KernelSnapShotPointer =
            | KernelSnapShotPointer;
20158|             B = DeviceIoControl(
            | ThreadStorage->PSManHandle,
20159|                 IOCTL_REVERT_TO_PRISTINE,
20160|                 &In,
20161|                 sizeof(In),
20162|                 NULL,
20163|                 0,
20164|                 &returned,
20165|                 &o);
20166|
20167|             if(B) {
20168|                 Err = 0;
20169|             } else {
20170|                 Err = GetLastError();
20171|                 DLOG((TEXT("Error %08x Calling
            | Psm_RevertSnapShotToPristineState\n"),Err));
20172|             }
20173|             CloseHandle(o.hEvent);
20174|         } else {
20175|             Err = ERROR_INVALID_HANDLE;
20176|         }

```

```

20177|     } __except(EXCEPTION_EXECUTE_HANDLER) {
20178|         Err = GetExceptionCode();
20179|         DLOG((TEXT("Exception %08x
| Psm_RevertSnapShotToPristineState\n"),Err));
20180|     }
20181|
20182|     return Err;
20183| }
20184|
20185| DLLEXPORT PSMSTATUS PSMAPI Psm_RevertToSnapShot (
20186|     PVOID     KernelSnapShotPointer,
20187|     ULONG     RevertFlags )
20188| {
20189|     ULONG B=FALSE;
20190|     LONG Err=0;
20191|     OVERLAPPED o;
20192|     ULONG returned=0;
20193|     pThreadStorage ThreadStorage = GetThreadStorage();
20194|     tPSM_RevertToSnapShotIn In = {0};
20195|
20196|     __try {
20197|         if
| ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
20198|             ULONG Cluster = ClusterIsClusterActive();
20199|
20200|             if(Cluster) {
20201|                 // or in At boot
20202|                 RevertFlags |= PSM_REVERT_FLAG_ATBOOT;
20203|             }
20204|
20205|             memset(&o,0,sizeof(o));
20206|             o.hEvent = CreateEvent( NULL, FALSE, FALSE,
| NULL );
20207|
20208|             In.KernelSnapShotPointer =
| KernelSnapShotPointer;
20209|             In.RevertFlags = RevertFlags;
20210|             B = DeviceIoControl(
| ThreadStorage->PSManHandle,
20211|                 IOCTL_REVERT_TO_SNAPSHOT,
20212|                 &In,
20213|                 sizeof(In),
20214|                 NULL,
20215|                 0,
20216|                 &returned,
20217|                 &o);
20218|
20219|             if(B) {
20220|                 Err = 0;
20221|             } else {

```



```

20222|         Err = GetLastError();
20223|         DLOG((TEXT("Error %08x Calling
| Psm_RevertToSnapShot\n"),Err));
20224|     }
20225|     CloseHandle(o.hEvent);
20226|
20227|     // initiate the revert by taking the volume
| offline and online
20228|     if((Err==PSM_ERROR_REBOOT_NEEDED) &&
| (Cluster)) {
20229|         Err =
| ClusterInitiateRevert(KernelSnapShotPointer);
20230|     }
20231|     } else {
20232|         Err = ERROR_INVALID_HANDLE;
20233|     }
20234| } __except(EXCEPTION_EXECUTE_HANDLER) {
20235|     Err = GetExceptionCode();
20236|     DLOG((TEXT("Exception %08x
| Psm_RevertToSnapShot\n"),Err));
20237| }
20238|
20239| return Err;
20240| }
20241|
20242|
20243| DLLEXPORT PSMSTATUS PSMAPI
| Psm_GetKernelSnapShotVolumesW (
20244|     PVOID KernelSnapShotPointer,
20245|     WCHAR *Buffer,
20246|     ULONG BufferSize )
20247| {
20248|     ULONG B=FALSE;
20249|     LONG Err=0;
20250|     OVERLAPPED o;
20251|     ULONG returned=0;
20252|     pThreadStorage ThreadStorage = GetThreadStorage();
20253|     tPSM_GetKernelSnapShotInfoIn In;
20254|     WCHAR *TempBuffer=LocalAlloc(LPTR,128*1024);
20255|     ULONG BytesLeft=BufferSize;
20256|
20257|     if(!TempBuffer) {
20258|         return ERROR_OUTOFMEMORY;
20259|     }
20260|
20261|     __try {
20262|         memset(Buffer,0,BufferSize);
20263|
20264|         if
| ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {

```

```

20265|         memset(&o,0,sizeof(o));
20266|         o.hEvent = CreateEvent( NULL, FALSE, FALSE,
| NULL );
20267|
20268|         In.KernelSnapShotPointer =
| KernelSnapShotPointer;
20269|
20270|         B = DeviceIoControl(
| ThreadStorage->PSManHandle,
20271|         IOCTL_GET_KERNEL_SNAPSHOT_VOLUMES,
20272|         &In,
20273|         sizeof(In),
20274|         TempBuffer,
20275|         128*1024,
20276|         &returned,
20277|         &o);
20278|
20279|         if(B) {
20280|             WCHAR *p = TempBuffer;
20281|             Err = 0;
20282|             while(*p) {
20283|                 WCHAR Temp[256];
20284|                 ULONG LenCharsIn = wcslen(p) + 1;
20285|                 Err =
| GetDriveLetterForNtDeviceName(p,Temp);
20286|                 if(Err) {
20287|                     Err =
| GetWin32NameForNtDeviceName(p,Temp);
20288|                 }
20289|                 if(!Err) {
20290|                     ULONG LenCharsOut =
| wcslen(Temp) + 1;
20291|                     ULONG LenBytesOut = LenCharsOut
| * sizeof(WCHAR);
20292|                     if ( BytesLeft < LenBytesOut )
| {
20293|                         Err = ERROR_MORE_DATA;
20294|                         break;
20295|                     }
20296|                     wcscpy ( Buffer, Temp );
20297|                     Buffer += LenCharsOut;
20298|                     p += LenCharsIn;
20299|                     BytesLeft -= LenBytesOut;
20300|                 } else {
20301|                     break;
20302|                 }
20303|             }
20304|             if ( !Err ) {
20305|                 if ( BytesLeft > 0 ) {
20306|                     *Buffer = L'\0'; // 2 null

```

```

    | terminators in a row indicate end-of-list
20307|         } else {
20308|             Err = ERROR_MORE_DATA;
20309|         }
20310|     }
20311| } else {
20312|     Err = GetLastError();
20313|     DLOG((TEXT("Error %08x Calling
    | Psm_GetKernelSnapShotVolumes\n"),Err));
20314| }
20315|     CloseHandle(o.hEvent);
20316| } else {
20317|     Err = ERROR_INVALID_HANDLE;
20318| }
20319|     LocalFree(TempBuffer);
20320| } __except(EXCEPTION_EXECUTE_HANDLER) {
20321|     Err = GetExceptionCode();
20322|     DLOG((TEXT("Exception %08x
    | Psm_GetKernelSnapShotVolumes\n"),Err));
20323| }
20324|
20325| return Err;
20326|
20327| }
20328|
20329| DLLEXPORT PSMSTATUS PSMAPI
    | Psm_GetKernelSnapShotVolumesA (
20330|     PVOID KernelSnapShotPointer,
20331|     CHAR *Buffer,
20332|     ULONG BufferSize )
20333| {
20334|     ULONG Err = 0;
20335|     WCHAR *BufferW =
    | LocalAlloc(LPTR,BufferSize*sizeof(WCHAR));
20336|     if(BufferW) {
20337|         ULONG BufferBytesLeft = BufferSize;
20338|         WCHAR *pIn = BufferW;
20339|         CHAR *pOut = Buffer;
20340|
20341|         Err = Psm_GetKernelSnapShotVolumesW (
20342|             KernelSnapShotPointer,
20343|             BufferW,
20344|             BufferSize*sizeof(WCHAR) );
20345|
20346|         if(!Err) {
20347|             while ( *pIn ) {
20348|                 ULONG LenChars = wcslen(pIn) + 1;
20349|                 if ( LenChars < BufferBytesLeft ) {
20350|                     CharToOemW ( pIn, pOut );
20351|                     pIn += LenChars;

```

```

20352|         pOut += LenChars;
20353|         BufferBytesLeft -= LenChars;
20354|     } else {
20355|         Err = ERROR_BUFFER_OVERFLOW;
20356|         break;
20357|     }
20358| }
20359| }
20360|
20361| if ( Err == 0 ) {
20362|     if ( BufferBytesLeft > 0 ) {
20363|         *pOut = '\0'; // double null
        | terminate to mark end of list
20364|     } else {
20365|         Err = ERROR_BUFFER_OVERFLOW;
20366|     }
20367| }
20368| LocalFree(BufferW);
20369| } else {
20370|     Err = ERROR_OUTOFMEMORY;
20371| }
20372| return Err;
20373| }
20374|
20375| ULONG MakeSnapShotDirectoryName( WCHAR *DataOut, WCHAR
        | *UserPattern, PVOID KernelSnapShotPointer )
20376| {
20377|     WCHAR *Buffer = DataOut;
20378|     ULONG Current = 0;
20379|     WCHAR *p = 0;
20380|     WCHAR Buf[20];
20381|     FILETIME NewFileTime;
20382|     SYSTEMTIME SystemTime;
20383|     OVERLAPPED o;
20384|     ULONG returned = 0;
20385|     pThreadStorage ThreadStorage = GetThreadStorage();
20386|     tPSM_GetKernelSnapShotInfoOut Out={0};
20387|     tPSM_GetKernelSnapShotInfoIn In={0};
20388|     WCHAR *Pattern;
20389|
20390|     if(wcsncmp(UserPattern,L"")==0) {
20391|         Pattern = SnapShotPattern;
20392|     } else {
20393|         Pattern = UserPattern;
20394|     }
20395|
20396|     memset(&o,0,sizeof(o));
20397|     o.hEvent = CreateEvent( NULL, FALSE, FALSE, NULL );
20398|
20399|     In.KernelSnapShotPointer = KernelSnapShotPointer;

```

```

20400|
20401|     if (!DeviceIoControl( ThreadStorage->PSManHandle,
20402|         IOCTL_GET_KERNEL_SNAPSHOT_INFO,
20403|         &In,
20404|         sizeof(In),
20405|         &Out,
20406|         sizeof(Out),
20407|         &returned,
20408|         &o)) {
20409|         CloseHandle(o.hEvent);
20410|         return GetLastError();
20411|     }
20412|
20413|     CloseHandle(o.hEvent);
20414|
20415|     | FileTimeToLocalFileTime((FILETIME*)&Out.SnapShotTime,&Ne
      | wFileTime);
20416|     FileTimeToSystemTime(&NewFileTime,&SystemTime);
20417|
20418|     wcsncpy(Buffer,SnapShotLocation);
20419|     wscat(Buffer,L"\\");
20420|     Current=wcslen(Buffer);
20421|
20422|     for(p=Pattern;*p;p++) {
20423|         switch (*p) {
20424|             // remove invalid characters
20425|             case L':' :
20426|                 *p = L'-';
20427|                 Buffer[Current++] = *p;
20428|                 break;
20429|             case L'>' :
20430|                 *p = L'}';
20431|                 Buffer[Current++] = *p;
20432|                 break;
20433|             case L'<' :
20434|                 *p = L'{' ;
20435|                 Buffer[Current++] = *p;
20436|                 break;
20437|             case L'/' :
20438|                 *p = L'-';
20439|                 Buffer[Current++] = *p;
20440|                 break;
20441|             case L'\\' :
20442|                 *p = L'-';
20443|                 Buffer[Current++] = *p;
20444|                 break;
20445|             case L'|' :
20446|                 *p = L'!' ;
20447|                 Buffer[Current++] = *p;

```

```

20448|         break;
20449|     case L'*' :
20450|         *p = L'+';
20451|         Buffer[Current++] = *p;
20452|         break;
20453|     case L'?' :
20454|         *p = L'.';
20455|         Buffer[Current++] = *p;
20456|         break;
20457|     case L'\" :
20458|         *p = L'\";
20459|         Buffer[Current++] = *p;
20460|         break;
20461|     // check for special chars
20462|     case L'%' :
20463|         // M = Month, D = Day, Y=Year
20464|         // n = 3 letter month name, N = Full
20465|         | month name
20466|         // m = minute, h=12 hour, H=24 hour,
20467|         | s=second,
20468|         // w = 3 letter day of the week, W =
20469|         | full day of week
20470|         // a = ampm
20471|         // p = private data
20472|         // i = instance
20473|         switch(*(p+1)) {
20474|             case L'%':
20475|                 wcscpy(Buf,L"%%%");
20476|                 break;
20477|             case L'M':
20478|                 | swprintf(Buf,L"%02d",SystemTime.wMonth);
20479|                 break;
20480|             case L'D':
20481|                 | swprintf(Buf,L"%02d",SystemTime.wDay);
20482|                 break;
20483|             case L'Y':
20484|                 | swprintf(Buf,L"%4d",SystemTime.wYear);
20485|                 break;
20486|             case L'm':
20487|                 | swprintf(Buf,L"%02d",SystemTime.wMinute);
20488|                 break;
20489|             case L'h':
20490|                 if(SystemTime.wHour>12) {
20491|                     | swprintf(Buf,L"%02d",SystemTime.wHour-12);
20492|                 } else {

```

```

20490|                if(SystemTime.wHour==0) {
20491|                    swprintf(Buf,L"12");
20492|                } else {
20493|
20494|                    | swprintf(Buf,L"%02d",SystemTime.wHour);
20495|                }
20496|            }
20497|            break;
20498|            case L'H':
20499|                | swprintf(Buf,L"%02d",SystemTime.wHour);
20500|            break;
20501|            case L's':
20502|                | swprintf(Buf,L"%02d",SystemTime.wSecond);
20503|            break;
20504|            case L'w': {
20505|                WCHAR *Days[] =
20506|                | {L"Sun",L"Mon",L"Tue",L"Wed",L"Thu",L"Fri",L"Sat"};
20507|                | swprintf(Buf,L"%s",Days[SystemTime.wDayOfWeek]);
20508|            break;
20509|        }
20510|        case L'W': {
20511|            WCHAR *Days[] =
20512|            | {L"Sunday",L"Monday",L"Tuesday",L"Wednesday",L"Thursday"
20513|            | ,L"Friday",L"Saturday"};
20514|            | swprintf(Buf,L"%s",Days[SystemTime.wDayOfWeek]);
20515|        break;
20516|        }
20517|        case L'n': {
20518|            WCHAR *Months[] =
20519|            | {L"Jan",L"Feb",L"Mar",L"Apr",L"May",L"Jun",L"Jul",L"Aug"
20520|            | ,L"Sep",L"Oct",L"Nov",L"Dec"};
20521|            | swprintf(Buf,L"%s",Months[SystemTime.wMonth-1]);
20522|        break;
20523|        }
20524|        case L'N': {
20525|            WCHAR *Months[] =
20526|            | {L"January",L"February",L"March",L"April",L"May",L"June"
20527|            | ,L"July",L"August",L"September",L"October",L"November",L
20528|            | "December"};
20529|            | swprintf(Buf,L"%s",Months[SystemTime.wMonth-1]);
20530|        break;
20531|        }
20532|        case L'a':
20533|            if(SystemTime.wHour>12) {

```

```

20525|             swprintf(Buf,L"PM");
20526|         } else {
20527|             swprintf(Buf,L"AM");
20528|         }
20529|         break;
20530|         case L'p':
20531|
20532|         | swprintf(Buf,L"%08x",Out.CallerPrivateUse);
20533|             break;
20534|         case L'P':
20535|
20536|         | swprintf(Buf,L"%08x",Out.Priority);
20537|             break;
20538|         case L'k':
20539|
20540|         | swprintf(Buf,L"%08x",Out.NumToKeep);
20541|             break;
20542|         default:
20543|             wcscpy(Buf,L "");
20544|             break;
20545|         }
20546|         // skip over character after percent
20547|         | sign
20548|             p++;
20549|             wcscpy(&Buffer[Current],Buf);
20550|             Current+=wcslen(Buf);
20551|             break;
20552|         default:
20553|             Buffer[Current++] = *p;
20554|             break;
20555|         }
20556|     }
20557|     Buffer[Current]=0;
20558|
20559|     // if you can read this, you are too close
20560|     return 0;
20561| }
20562|
20563|
20564| DLLEXPORT PSMSTATUS PSMAPI Psm_GetKernelSnapShotInfoW(
20565|     | PVOID KernelSnapShotPointer,
20566|     | tPSM_GetKernelSnapShotInfoW *Info )
20567| {
20568|     ULONG B=FALSE;
20569|     LONG Err=0;

```



```

20568| OVERLAPPED o;
20569| ULONG returned=0;
20570| pThreadStorage ThreadStorage = GetThreadStorage();
20571| tPSM_GetKernelSnapShotInfoOut Out;
20572| tPSM_GetKernelSnapShotInfoIn In;
20573|
20574| __try {
20575|     if
        | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
20576|         memset(&o,0,sizeof(o));
20577|         o.hEvent = CreateEvent( NULL, FALSE, FALSE,
        | NULL );
20578|
20579|         In.KernelSnapShotPointer =
        | KernelSnapShotPointer;
20580|
20581|         B = DeviceIoControl(
        | ThreadStorage->PSManHandle,
20582|         IOCTL_GET_KERNEL_SNAPSHOT_INFO,
20583|         &In,
20584|         sizeof(In),
20585|         &Out,
20586|         sizeof(Out),
20587|         &returned,
20588|         &o);
20589|
20590|         if(B) {
20591|             Err = 0;
20592|             Info->CallerPrivateUse =
        | Out.CallerPrivateUse;
20593|             Info->NumToKeep = Out.NumToKeep;
20594|             Info->Priority = Out.Priority;
20595|             Info->SnapShotFlags =
        | Out.SnapShotFlags;
20596|             Info->Instance = Out.Instance;
20597|             Info->Persistent = Out.Persistent;
20598|             Info->SnapShotTime = Out.SnapShotTime;
20599|             Info->Status = Out.Status;
20600|             wcscpy(Info->Directory,L"\");
20601|
        | wcscpy(Info->Directory,Out.UserSnapShotName);
20602|
        | //MakeSnapShotDirectoryName(&Info->Directory[1],
        | KernelSnapShotPointer );
20603|
20604|         } else {
20605|             Err = GetLastError();
20606|             DLOG((TEXT("Error %08x Calling
        | Psm_GetKernelSnapShotInfo\n"),Err));
20607|         }

```

```

20608|         CloseHandle(o.hEvent);
20609|     } else {
20610|         Err = ERROR_INVALID_HANDLE;
20611|     }
20612| } __except(EXCEPTION_EXECUTE_HANDLER) {
20613|     Err = GetExceptionCode();
20614|     DLOG((TEXT("Exception %08x
| Psm_GetKernelSnapShotInfo\n"),Err));
20615| }
20616|
20617| return Err;
20618| }
20619|
20620| DLLEXPORT PSMSTATUS PSMAPI Psm_GetKernelSnapShotInfoA(
| PVOID KernelSnapShotPointer,
| tPSM_GetKernelSnapShotInfoA *Info )
20621| {
20622|     ULONG Err;
20623|     tPSM_GetKernelSnapShotInfoW InfoW;
20624|
20625|     Err =
| Psm_GetKernelSnapShotInfoW(KernelSnapShotPointer,&InfoW)
| ;
20626|
20627|     if(!Err) {
20628|         CharToOemW(InfoW.Directory,Info->Directory);
20629|
20630|         Info->CallerPrivateUse =
| InfoW.CallerPrivateUse;
20631|         Info->NumToKeep = InfoW.NumToKeep;
20632|         Info->Priority = InfoW.Priority;
20633|         Info->SnapShotFlags = InfoW.SnapShotFlags;
20634|         Info->Instance = InfoW.Instance;
20635|         Info->Persistent = InfoW.Persistent;
20636|         Info->SnapShotTime = InfoW.SnapShotTime;
20637|         Info->Status = InfoW.Status;
20638|     }
20639|
20640|     return Err;
20641| }
20642|
20643| DLLEXPORT PSMSTATUS PSMAPI Psm_SetKernelSnapShotInfoW(
| PVOID KernelSnapShotPointer, tPSM_SetKernelSnapShotInfo
| *Info )
20644| {
20645|     ULONG B=FALSE;
20646|     LONG Err;
20647|     OVERLAPPED o;
20648|     ULONG returned;
20649|     pThreadStorage ThreadStorage = GetThreadStorage();

```

```

20650|   tPSM_SetKernelSnapShotInfoIn In;
20651|
20652|   __try {
20653|       if
20654|           | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
20655|               memset(&o,0,sizeof(o));
20656|               o.hEvent = CreateEvent( NULL, FALSE, FALSE,
20657|                   | NULL );
20658|               In.KernelSnapShotPointer =
20659|                   | KernelSnapShotPointer;
20660|               memcpy(&In.Info,Info,sizeof(tPSM_SetKernelSnapShotInfo))
20661|                   | ;
20662|               B = DeviceIoControl(
20663|                   | ThreadStorage->PSManHandle,
20664|                   IOCTL_SET_KERNEL_SNAPSHOT_INFO,
20665|                   &In,
20666|                   sizeof(In),
20667|                   NULL,
20668|                   0,
20669|                   &returned,
20670|                   &o);
20671|               if(B) {
20672|                   Err = 0;
20673|               } else {
20674|                   Err = GetLastError();
20675|                   DLOG((TEXT("Error %08x Calling
20676|                       | Psm_SetKernelSnapShotInfo\n"),Err));
20677|               }
20678|               CloseHandle(o.hEvent);
20679|           } else {
20680|               Err = ERROR_INVALID_HANDLE;
20681|           }
20682|       } __except(EXCEPTION_EXECUTE_HANDLER) {
20683|           Err = GetExceptionCode();
20684|           DLOG((TEXT("Exception %08x
20685|               | Psm_SetKernelSnapShotInfo\n"),Err));
20686|       }
20687|   }
20688|   return Err;
20689| }
20690|
20691| DLLEXPORT PSMSTATUS PSMAPI Psm_GetVolumeCacheInfoW(
20692|     | WCHAR *VolumeName, pPSM_GetVolumeCacheInfoOut Out );
20693|
20694| ULONGLONG GetSizeOfCacheFileFromReg( WCHAR
20695|     | *NtDeviceName, WCHAR *Guid, WCHAR *Unique)

```

```

20690| {
20691|     ULONG DataSize;
20692|     HKEY Key;
20693|     ULONG Err;
20694|     WCHAR RegStr[256];
20695|     ULONG InitialSize=-1;
20696|     PPARTITION_INFORMATION PartInfo;
20697|
20698|     // lets check the registry for this file
20699|
20700|
        | swprintf(RegStr,L"SYSTEM\\CurrentControlSet\\Services\\P
        | SMan%d\\", NtBuildNumber > 1381 ? 5 : 4);
20701|     // guid = "\\??\volume{...}"
20702|     wcscat(RegStr,Guid+11);
20703|     // get rid of last curly braket
20704|     RegStr[wcslen(RegStr)-1]=0;
20705|
20706|
20707|     // open the registry
20708|     Err = RegOpenKeyExW(
20709|         HKEY_LOCAL_MACHINE, // handle of open key
20710|         RegStr, // address of name of subkey to open
20711|         0, // reserved
20712|         KEY_READ, // security access mask
20713|         &Key// address of handle of open key
20714|     );
20715|
20716|     if(Err==0) {
20717|         DataSize = 4;
20718|
20719|         Err =
            | RegQueryValueExW(Key,L"InitialSize",NULL,NULL,(char*)&In
            | itialSize,&DataSize );
20720|         if(Err) {
20721|             DLOG((TEXT("GetSizeOfCacheFileFromReg: no
            | volume specific value\n")));
20722|             goto Default;
20723|         }
20724|
20725|         // close the registry
20726|         RegCloseKey(Key);
20727|     } else {
20728|         DLOG((TEXT("GetSizeOfCacheFileFromReg: no
            | volume specific key\n")));
20729| Default:
20730|         // key doesnt exist try default
20731|
        | swprintf(RegStr,L"SYSTEM\\CurrentControlSet\\Services\\P
        | SMan%d\\Persistent", NtBuildNumber > 1381 ? 5 : 4);

```

```

20732|     Err = RegOpenKeyExW(
20733|         HKEY_LOCAL_MACHINE, // handle of open key
20734|         RegStr, // address of name of subkey to
        | open
20735|         0, // reserved
20736|         KEY_READ, // security access mask
20737|         &Key// address of handle of open key
20738|     );
20739|
20740|     if(Err==0) {
20741|         DataSize = 4;
20742|
20743|         Err =
        | RegQueryValueExW(Key,L"d_InitialSize",NULL,NULL,(char*)&
        | InitialSize,&DataSize );
20744|         if(Err) {
20745|             DLOG((TEXT("GetSizeOfCacheFileFromReg:
        | no default value\n"))));
20746|             // set up default...
20747|             InitialSize = -20;
20748|         }
20749|
20750|         // close the registry
20751|         RegCloseKey(Key);
20752|     } else {
20753|         DLOG((TEXT("GetSizeOfCacheFileFromReg: no
        | default key\n"))));
20754|         InitialSize = -20;
20755|     }
20756| }
20757| ASSERT(InitialSize!=0);
20758|
20759| if((signed)InitialSize<0) {
20760|     if((signed)InitialSize>-100) {
20761|         // okay cache allocation is based on
        | percentage of disk
20762|
20763|         PartInfo = LocalAlloc(LPTR,8192);
20764|         if(PartInfo) {
20765|             UNICODE_STRING Uni;
20766|             OBJECT_ATTRIBUTES ObjectAttributes;
20767|             HANDLE hVolume;
20768|             IO_STATUS_BLOCK IoStatus;
20769|             ULONG dwBytesReturned;
20770|
20771|
20772|             RtlInitUnicodeString( &Uni,
        | NtDeviceName);
20773|
20774|             InitializeObjectAttributes (

```

```

    | &ObjectAttributes,
20775|                                     &Uni,
20776|
    | OBJ_CASE_INSENSITIVE,
20777|                                     NULL,
20778|                                     NULL );
20779|
20780|     Err = NtCreateFile( &hVolume,
20781|
    | FILE_GENERIC_READ,           // desired access
20782|
    | &ObjectAttributes,           // object attributes
20783|                                     &IoStatus,
20784|                                     NULL,
    | // alloc size
20785|
    | FILE_ATTRIBUTE_NORMAL,       // file attributes
20786|                                     FILE_SHARE_WRITE
    | | FILE_SHARE_READ,           // share
    | access
20787|                                     FILE_OPEN,
    | // create disposition
20788|
    | FILE_SYNCHRONOUS_IO_NONALERT, // create
    | options
20789|                                     NULL, //
    | eabuffer
20790|                                     0 ); //
    | ealength
20791|
20792|     if(!Err) {
20793|         if(DeviceIoControl(hVolume,
20794|             IOCTL_DISK_GET_PARTITION_INFO,
20795|             NULL, 0,
20796|             PartInfo, 8192,
20797|             &dwBytesReturned,
20798|             NULL)) {
20799|             LARGE_INTEGER L;
20800|             ULONG ISC;
20801|
20802|             // get it in MB
20803|
    | L.QuadPart=PartInfo->PartitionLength.QuadPart /
    | (1024*1024);
20804|             ASSERT(L.HighPart == 0);
20805|             ISC = (L.LowPart *
    | abs(InitialSize)) / 100;
20806|             // return number of bytes
20807|             InitialSize = ISC;
20808|

```

```

    | DLOG((TEXT("GetSizeOfCacheFileFromReg: Calculated
    | InitialSize=%d MB (vol=%l64d
    | MB)\n"),InitialSize,L.QuadPart));
20809|         } else {
20810|             // cant get size of volume
20811|             DLOG((TEXT("Error %08x getting
    | partition info\n"),Err));
20812|             InitialSize = 20;
20813|         }
20814|         NtClose(hVolume);
20815|     } else {
20816|         // cant open volume to get size of
    | it.
20817|         DLOG((TEXT("Error %08x opening
    | volume\n"),Err));
20818|         InitialSize = 20;
20819|     }
20820|
20821|     LocalFree(PartInfo);
20822| } else {
20823|     DLOG((TEXT("GetSizeOfCacheFileFromReg:
    | out of memory\n")));
20824|     // cant alloc memory..
20825|     InitialSize = 20;
20826| }
20827| } else {
20828|     DLOG((TEXT("GetSizeOfCacheFileFromReg:
    | Number is negative but above 100%%!!!!\n")));
20829|     // leave InitialSize as what was read from
    | registry
20830| }
20831| } else {
20832|     DLOG((TEXT("GetSizeOfCacheFileFromReg:
    | Specified InitialSize=%d MB\n"),InitialSize));
20833|     // leave InitialSize as what was read from
    | registry
20834| }
20835|
20836| return (ULONGLONG)InitialSize*1024*1024;
20837| }
20838|
20839| #if 0
20840| // this is from an aborted attempt to create the cache
    | file fast.
20841| // This code can be removed later on
20842| ULONG CreatePSMWorkDirectory ( WCHAR *Root );
20843| ULONG TryCreateFilesUserMode(pOpenTransactionInInternal
    | Internal)
20844| {
20845|     ULONG i;

```

```

20846|    ULONG Err=0;
20847|
20848|    for(i=0;i<Internal->NumberOfDevices;i++) {
20849|        WCHAR Guid[256]=L"\\??\\";
20850|        WCHAR Unique[256];
20851|
20852|        {
20853|            WCHAR Dir[512];
20854|
20855|
20856|            | if(GetDriveLetterForNtDeviceName(DN_MakePointer(Internal
20857|            | ,Internal->DeviceName[i]), Unique)!=0) {
20858|                GetWin32NameForNtDeviceName(
20859|                | DN_MakePointer(Internal,Internal->DeviceName[i]),
20860|                | Unique);
20861|            }
20862|
20863|            swprintf(Dir,L"%s\\Persistent Storage
20864|            | Manager State",Unique);
20865|            // make root dir for snapshots on this
20866|            | volume if it
20867|            // doesnt exist
20868|            CreatePSMWorkDirectory(Dir);
20869|        }
20870|
20871|        | if(GetVolumeGuidForNtDeviceName(DN_MakePointer(Internal,
20872|        | Internal->DeviceName[i]),Guid+4)==0) {
20873|            if(Psm_GetVolumeUIDW(Guid,Unique)==0) {
20874|                WCHAR File1[256];
20875|                WCHAR File2[256];
20876|                WCHAR File3[256];
20877|                tNTFS_AllocFiles Files[3];
20878|                LONGLONG CacheSize;
20879|
20880|                Files[0].FileNameOnly = File1;
20881|                Files[1].FileNameOnly = File2;
20882|                Files[2].FileNameOnly = File3;
20883|
20884|                CacheSize = GetSizeOfCacheFileFromReg(
20885|                | DN_MakePointer(Internal,Internal->DeviceName[i]),Guid,Un
20886|                | ique);
20887|                // this is hardcoded to be 2 megs
20888|                | exactly
20889|                Files[0].FileSize.QuadPart =
20890|                | 2*1024*1024;
20891|                Files[1].FileSize.QuadPart = CacheSize
20892|                | / 128;
20893|                Files[2].FileSize.QuadPart = CacheSize;
20894|

```



```

20883|         DLOG((TEXT("NT='%ws', guid='%ws',
| unique='%ws\\n"),DN_MakePointer(Internal,Internal->Devic
| eName[i]),Guid,Unique));
20884|         DLOG((TEXT(" Cache size = %!64d
| MB\\n"),CacheSize / 1024 / 1024));
20885|
20886|         swprintf(File1,L"\\Persistent Storage
| Manager State\\%s.header.psm",Unique);
20887|         swprintf(File2,L"\\Persistent Storage
| Manager State\\%s.index.psm",Unique);
20888|         swprintf(File3,L"\\Persistent Storage
| Manager State\\%s.diff.psm",Unique);
20889|
20890|         Err =
| NTFS_AllocAndInitFiles(DN_MakePointer(Internal,Internal-
| >DeviceName[i]),3,Files);
20891|     } else {
20892|         DLOG((TEXT("unable to retrieve unique
| id\\n")));
20893|         Err=ERROR_WMI_GUID_NOT_FOUND;
20894|     }
20895| } else {
20896|     DLOG((TEXT("unable to retrieve volume
| guid\\n")));
20897|     Err=ERROR_WMI_GUID_NOT_FOUND;
20898| }
20899| }
20900| return Err;
20901| }
20902| #endif
20903|
20904|
20905| DLLEXPORT PSMSTATUS PSMAPI Psm_CreateFilesW( ULONG
| NumVolumes, WCHAR **InVolumeMap, HANDLE AbortEvent )
20906| {
20907|     ULONG Err;
20908|     // 4 bytes per pointer, 100 bytes (by default) of
| "string space"
20909|     ULONG InternalBufferChars = 100 * NumVolumes;
20910|     pOpenTransactionInInternal Internal;
20911|     ULONG
| Size=sizeof(tOpenTransactionInInternal)+(NumVolumes*4)+(
| InternalBufferChars*sizeof(WCHAR));
20912|     pThreadStorage ThreadStorage = GetThreadStorage();
20913|
20914|     Internal = LocalAlloc(LPTR,Size);
20915|
20916|     if(Internal) {
20917|         Err = MakeVolumeList (
20918|             NumVolumes,

```

```

20919|         InVolumeMap,
20920|         Size,
20921|         Internal,
20922|         InternalBufferChars );
20923|
20924|         if(Err==0) {
20925|             Internal->Size = sizeof(*Internal);
20926|             Internal->AbortEvent = AbortEvent;
20927|
20928| #if 0
20929|             // see comment for this function for reason
20930|             | it is commented out
20931|             // try and create the files in user mode.
20932|             // it doesnt matter if this routine
20933|             | succeeds or not as
20934|             // the kernel mode will fix everything up
20935|             | (or leave it alone,
20936|             // if it actually succeeds) We do this so
20937|             | we can create the
20938|             // cache files really fast
20939|             TryCreateFilesUserMode(Internal);
20940| #endif
20941|         __try {
20942|             if
20943|             | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
20944|                 ULONG B=FALSE;
20945|                 OVERLAPPED o;
20946|                 ULONG returned;
20947|                 memset(&o,0,sizeof(o));
20948|                 o.hEvent = CreateEvent( NULL,
20949|                 | FALSE, FALSE, NULL );
20950|
20951|                 B = DeviceIoControl(
20952|                 | ThreadStorage->PSManHandle,
20953|                 IOCTL_CREATE_FILES,
20954|                 Internal,
20955|                 Size,
20956|                 NULL,
20957|                 0,
20958|                 &returned,
20959|                 &o);
20960|
20961|                 if(B) {
20962|                     Err = 0;
20963|                 } else {
20964|                     Err = GetLastError();
20965|                     DLOG((TEXT("Error %08x Calling
20966|                     | Psm_GetVolumeInfoW\n"),Err));

```

```

20961|         }
20962|         CloseHandle(o.hEvent);
20963|     } else {
20964|         Err = ERROR_INVALID_HANDLE;
20965|     }
20966| } __except(EXCEPTION_EXECUTE_HANDLER) {
20967|     Err = GetExceptionCode();
20968|     DLOG((TEXT("Exception %08x
| Psm_GetVolumeInfoW\n"),Err));
20969| }
20970|
20971| } else {
20972|     DLOG((TEXT("Error %08x making volume
| list\n"),Err));
20973| }
20974|
20975|     LocalFree(Internal);
20976| } else {
20977|     Err = ERROR_OUTOFMEMORY;
20978| }
20979| return Err;
20980| }
20981|
20982| DLLEXPORT PSMSTATUS PSMAPI Psm_CreateFilesA( ULONG
| NumVolumes, CHAR **InVolumeMap, HANDLE AbortEvent )
20983| {
20984|     PWCHAR *Vmw=NULL;
20985|     ULONG Err=0;
20986|     ULONG i;
20987|     CHAR **InV;
20988|
20989|     __try {
20990|         (PVOID)InV = InVolumeMap;
20991|
20992|         Vmw =
| LocalAlloc(LPTR,NumVolumes*sizeof(PVOID));
20993|         if(!Vmw) {
20994|             try_return(Err=ERROR_OUTOFMEMORY);
20995|         }
20996|
20997|         for(i=0;i<NumVolumes;i++) {
20998|             Vmw[i] =
| LocalAlloc(LPTR,strlen(InV[i])*sizeof(WCHAR)+sizeof(WCHA
| R));
20999|             if(!Vmw[i]) {
21000|                 ULONG j;
21001|                 for (j=0;j<i;j++) {
21002|                     LocalFree(Vmw[j]);
21003|                 }
21004|                 LocalFree(Vmw);

```

```

21005|         try_return(Err=ERROR_OUTOFMEMORY);
21006|     }
21007|     OemToCharW(InV[i],Vmw[i]);
21008| }
21009|     Err =
    | Psm_CreateFilesW(NumVolumes,Vmw,AbortEvent);
21010|     for(i=0;i<NumVolumes;i++) {
21011|         LocalFree(Vmw[i]);
21012|         Vmw[i] = NULL;
21013|     }
21014|     LocalFree(Vmw);
21015| try_exit:
21016|     // need to have a statement after a label
21017|     ;
21018| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
21019|     Err = GetExceptionCode();
21020|     DLOG((TEXT("Exception %08x converting ansi to
    | wide during open\n"),Err));
21021| }
21022| return Err;
21023| }
21024|
21025| DLLEXPORT PSMSTATUS PSMAPI Psm_GetVolumeCacheInfoW(
    | WCHAR *VolumeName, pPSM_GetVolumeCacheInfoOut Out )
21026| {
21027|     ULONG B=FALSE;
21028|     LONG Err;
21029|     OVERLAPPED o;
21030|     ULONG returned;
21031|     pThreadStorage ThreadStorage = GetThreadStorage();
21032|     tPSM_GetVolumeCacheInfoIn In;
21033|
21034|     __try {
21035|         if
    | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
21036|             memset(&o,0,sizeof(o));
21037|             o.hEvent = CreateEvent( NULL, FALSE, FALSE,
    | NULL );
21038|
21039|
    | Err=GetNTDeviceName(VolumeName,In.VolumeName,256,FALSE);
21040|
21041|             if(!Err) {
21042|                 B = DeviceIoControl(
    | ThreadStorage->PSManHandle,
21043|                 IOCTL_GET_VOLUME_CACHE_INFO,
21044|                 &In,
21045|                 sizeof(In),
21046|                 Out,

```

```

21047|         sizeof(*Out),
21048|         &returned,
21049|         &o);
21050|
21051|         if(B) {
21052|             Err = 0;
21053|         } else {
21054|             Err = GetLastError();
21055|             DLOG((TEXT("Error %08x Calling
| Psm_GetVolumeInfoW\n"),Err));
21056|         }
21057|     }
21058|     CloseHandle(o.hEvent);
21059| } else {
21060|     Err = ERROR_INVALID_HANDLE;
21061| }
21062| } __except(EXCEPTION_EXECUTE_HANDLER) {
21063|     Err = GetExceptionCode();
21064|     DLOG((TEXT("Exception %08x
| Psm_GetVolumeInfoW\n"),Err));
21065| }
21066|
21067| return Err;
21068|
21069| }
21070|
21071| DLLEXPORT PSMSTATUS PSMAPI Psm_GetVolumeCacheInfoA(
| CHAR *VolumeName, pPSM_GetVolumeCacheInfoOut Out )
21072| {
21073|     WCHAR VolumeNameW[256];
21074|     OemToCharW(VolumeName, VolumeNameW);
21075|     return Psm_GetVolumeCacheInfoW(VolumeNameW, Out);
21076| }
21077|
21078|
21079| DLLEXPORT PSMSTATUS PSMAPI
| Psm_GetSnapshotInfoFromVolumeW( WCHAR *VolumeName,
| tSnapshotInfoW *SnapshotInfo )
21080| {
21081|     return
| PSMI_GetSnapshotInfoFromVolume(VolumeName, SnapshotInfo);
21082| }
21083|
21084| DLLEXPORT PSMSTATUS PSMAPI
| Psm_GetSnapshotInfoFromVolumeA( CHAR *VolumeName,
| tSnapshotInfoA *SnapshotInfo )
21085| {
21086|     WCHAR *VolumeNameW;
21087|     tSnapshotInfoW SSIW;
21088|     ULONG Err=0;

```

```

21089|
21090|     if(VolumeName) {
21091|         VolumeNameW =
            | LocalAlloc(LPTR,strlen(VolumeName)*sizeof(WCHAR)+sizeof(
            | WCHAR));
21092|         if(VolumeNameW) {
21093|             OemToCharW(VolumeName,VolumeNameW);
21094|             SSIW.Size = sizeof(tSnapshotInfoW);
21095|
21096|             Err =
            | Psm_GetSnapshotInfoFromVolumeW(VolumeNameW,&SSIW);
21097|             if(!Err) {
21098|
            | memmove(&(SnapshotInfo->Snapshot),&(SSIW.Snapshot),sizeo
            | f(tSnapshot));
21099|
            | CharToOemW(SSIW.Company,SnapshotInfo->Company);
21100|
            | CharToOemW(SSIW.Product,SnapshotInfo->Product);
21101|
            | CharToOemW(SSIW.Version,SnapshotInfo->Version);
21102|
            | CharToOemW(SSIW.Code,SnapshotInfo->Code);
21103|             CharToOemW(SSIW.Key,SnapshotInfo->Key);
21104|
            | CharToOemW(SSIW.Volume,SnapshotInfo->Volume);
21105|         }
21106|         LocalFree(VolumeNameW);
21107|     } else {
21108|         Err = ERROR_OUTOFMEMORY;
21109|     }
21110| } else {
21111|     Err = ERROR_INVALID_PARAMETER;
21112| }
21113| return Err;
21114| }
21115|
21116| DLLEXPORT PSMSTATUS PSMAPI Psm_IsAnPSMVolumeW( WCHAR
            | *VolumeName )
21117| {
21118|     return PSMI_IsAnPSMVolume(VolumeName);
21119| }
21120|
21121| DLLEXPORT PSMSTATUS PSMAPI Psm_IsAnPSMVolumeA( CHAR
            | *VolumeName )
21122| {
21123|     ULONG Err;
21124|     WCHAR
            | *VolNameW=LocalAlloc(LPTR,(strlen(VolumeName)+1) *
            | sizeof(WCHAR));

```

```

21125|   if(VolNameW) {
21126|       OemToCharW(VolumeName,VolNameW);
21127|       Err = Psm_IsAnPSMVolumeW(VolNameW);
21128|       LocalFree(VolNameW);
21129|   } else {
21130|       Err = ERROR_OUTOFMEMORY;
21131|   }
21132|   return Err;
21133| }
21134|
21135| DLLEXPORT PSMSTATUS PSMAPI Psm_GetVolumeStatsW( WCHAR
    | *VolumeName, tPSM_GetStatsRecord *Stats )
21136| {
21137|   pThreadStorage ThreadStorage = GetThreadStorage();
21138|
21139|   if
    | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
    | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {
21140|       return
    | PSMI_GetVolumeStats(ThreadStorage->PSManHandle,ThreadSto
    | rage->PSManEvent,VolumeName,Stats);
21141|   } else {
21142|       return ERROR_INVALID_HANDLE;
21143|   }
21144| }
21145|
21146| DLLEXPORT PSMSTATUS PSMAPI Psm_GetVolumeStatsA( CHAR
    | *VolumeName, tPSM_GetStatsRecord *Stats )
21147| {
21148|   ULONG Err;
21149|   WCHAR
    | *VolNameW=LocalAlloc(LPTR,(strlen(VolumeName)+1) *
    | sizeof(WCHAR));
21150|   if(VolNameW) {
21151|       OemToCharW(VolumeName,VolNameW);
21152|       Err = Psm_GetVolumeStatsW(VolNameW,Stats);
21153|       LocalFree(VolNameW);
21154|   } else {
21155|       Err = ERROR_OUTOFMEMORY;
21156|   }
21157|   return Err;
21158| }
21159|
21160| DLLEXPORT PSMSTATUS PSMAPI Psm_CanBePSMedW( WCHAR
    | *VolumeName )
21161| {
21162|   ULONG VolumeType;
21163|   return PSMI_CanBePSMed(VolumeName,&VolumeType);
21164| }
21165|

```

```

21166| DLLEXPORT PSMSTATUS PSMAPI Psm_CanBePSMedA( CHAR
      | *VolumeName )
21167| {
21168|     ULONG Err;
21169|     WCHAR
      | *VolNameW=LocalAlloc(LPTR,(strlen(VolumeName)+1) *
      | sizeof(WCHAR));
21170|     if(VolNameW) {
21171|         OemToCharW(VolumeName,VolNameW);
21172|         Err = Psm_CanBePSMedW(VolNameW);
21173|         LocalFree(VolNameW);
21174|     } else {
21175|         Err = ERROR_OUTOFMEMORY;
21176|     }
21177|     return Err;
21178| }
21179|
21180| DLLEXPORT PSMSTATUS PSMAPI Psm_IsSupportedTypeW( WCHAR
      | *VolumeName )
21181| {
21182|     ULONG VolumeType=PSM_IS_UNKNOWN;
21183|     PSMI_CanBePSMed(VolumeName,&VolumeType);
21184|     return VolumeType;
21185| }
21186|
21187| DLLEXPORT PSMSTATUS PSMAPI Psm_IsSupportedTypeA( CHAR
      | *VolumeName )
21188| {
21189|     ULONG Type;
21190|     WCHAR
      | *VolNameW=LocalAlloc(LPTR,(strlen(VolumeName)+1) *
      | sizeof(WCHAR));
21191|     if(VolNameW) {
21192|         OemToCharW(VolumeName,VolNameW);
21193|         Type = Psm_IsSupportedTypeW(VolNameW);
21194|         LocalFree(VolNameW);
21195|     } else {
21196|         Type = PSM_IS_UNKNOWN;
21197|     }
21198|     return Type;
21199| }
21200|
21201| DLLEXPORT PSMSTATUS PSMAPI Psm_FreeFilesW( IN pSnapShot
      | SnapShot, IN ULONG NumberOfFiles, WCHAR *Files[])
21202| {
21203|     ULONG Err=0;
21204|     pThreadStorage ThreadStorage = GetThreadStorage();
21205|
21206|     __try {
21207|         //DLOG(TEXT("Psm_FreeFiles Called

```



```

    | %d\n"),NumberOfFiles));
21208|
21209|     if ( (NumberOfFiles>0) &&
21210|
    | (ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
21211|
    | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE))
21212|         Err =
    | PSMI_FreeFiles(ThreadStorage->PSManHandle,ThreadStorage-
    | >PSManEvent,SnapShot,NumberOfFiles,Files);
21213|
21214|     } __except(EXCEPTION_EXECUTE_HANDLER) {
21215|         Err = GetExceptionCode();
21216|         DLOG((TEXT("Exception %08x Freeing
    | Files\n"),Err));
21217|     }
21218|
21219|     return Err;
21220| }
21221|
21222| DLLEXPORT PSMSTATUS PSMAPI Psm_FreeRangesW( IN
    | tPSM_FreeRanges *Ranges )
21223| {
21224|     ULONG Err=0;
21225|     pThreadStorage ThreadStorage = GetThreadStorage();
21226|
21227|     __try {
21228| //         DLOG((TEXT("Psm_FreeRanges Called\n")));
21229|
21230|         if ( (Ranges->NumberOfRanges>0) &&
21231|
    | (ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
21232|
    | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE))
21233|             Err =
    | PSMI_FreeRanges(ThreadStorage->PSManHandle,ThreadStorage
    | ->PSManEvent,Ranges);
21234|
21235|     } __except(EXCEPTION_EXECUTE_HANDLER) {
21236|         DLOG((TEXT("Exception %08x Freeing
    | ranges\n"),Err));
21237|         Err = GetExceptionCode();
21238|     }
21239|
21240|     return Err;
21241| }
21242|
21243| DLLEXPORT PSMSTATUS PSMAPI Psm_FreeVolumeW( IN
    | pSnapShot SnapShot, IN WCHAR *VolumeName )
21244| {

```

```

21245|  ULONG Err=ERROR_INVALID_HANDLE;
21246|  pThreadStorage ThreadStorage = GetThreadStorage();
21247|  tPSM_FreeVolume Volume;
21248|
21249|  DLOG((TEXT("Psm_FreeVolume Called\n")));
21250|
21251|  __try {
21252|      if(!(Err =
        | GetNTDeviceName(VolumeName,Volume.VolumeName,256,FALSE))
        | ) {
21253|          if(!(Err = FreeOneVolumeForSnapShot (
        | SnapShot, Volume.VolumeName ))) {
21254|              Volume.KernelSnapShotPointer =
        | SnapShot->KernelSnapShotPointer;
21255|              if
        | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
        | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {
21256|                  Err=PSMI_FreeVolume(
        | ThreadStorage->PSManHandle, ThreadStorage->PSManEvent,
        | &Volume );
21257|              } else {
21258|                  DLOG((TEXT("Error %08x psm not
        | installed\n"),Err));
21259|              }
21260|              } else {
21261|                  DLOG((TEXT("Error %08x freeing volume
        | '%S' = '%S'\n"),Err,VolumeName,Volume.VolumeName));
21262|              }
21263|              } else {
21264|                  DLOG((TEXT("Error %08x getting device name
        | for '%S'\n"),Err,VolumeName));
21265|              }
21266|      } __except(EXCEPTION_EXECUTE_HANDLER) {
21267|          DLOG((TEXT("Exception %08x Freeing
        | Volume\n"),Err));
21268|          Err = GetExceptionCode();
21269|      }
21270|
21271|  return Err;
21272| }
21273|
21274| DLLEXPORT PSMSTATUS PSMAPI Psm_FreeVolumeA( IN
        | pSnapShot SnapShot, IN CHAR *VolumeName )
21275| {
21276|     WCHAR VolNameW[256];
21277|     OemToCharW(VolumeName,VolNameW);
21278|     return Psm_FreeVolumeW(SnapShot,VolNameW);
21279| }
21280|
21281| DLLEXPORT PSMSTATUS PSMAPI Psm_GetError( OUT

```

```

    | pPSM_GetErrorOut Out )
21282| {
21283|     ULONG Err=ERROR_INVALID_HANDLE;
21284|     pThreadStorage ThreadStorage = GetThreadStorage();
21285|
21286|     __try {
21287|         DLOG((TEXT("Psm_GetError Called\n")));
21288|         if
            | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
            | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {
21289|             Err=PSMI_GetError(
            | ThreadStorage->PSManHandle, ThreadStorage->PSManEvent,
            | Out, NULL);
21290|         } else {
21291|             DLOG((TEXT("Error %08x psm not
            | installed\n"),Err));
21292|         }
21293|     }
        | __except(ExceptionFilter(GetExceptionInformation())) {
21294|         Err = GetExceptionCode();
21295|         DLOG((TEXT("Exception %08x getting error
            | info\n"),Err));
21296|     }
21297|     return Err;
21298| }
21299|
21300| DLLEXPORT PSMSTATUS PSMAPI Psm_GetSnapShotError( IN
    | pSnapShot SnapShot, OUT pPSM_GetErrorOut Out )
21301| {
21302|     ULONG Err=ERROR_INVALID_HANDLE;
21303|     pThreadStorage ThreadStorage = GetThreadStorage();
21304|
21305|     __try {
21306|         DLOG((TEXT("Psm_GetSnapShotError Called\n")));
21307|
21308|         if
            | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
            | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {
21309|             Err=PSMI_GetError(
            | ThreadStorage->PSManHandle, ThreadStorage->PSManEvent,
            | Out, SnapShot );
21310|         } else {
21311|             DLOG((TEXT("Error %08x psm not
            | installed\n"),Err));
21312|         }
21313|     }
        | __except(ExceptionFilter(GetExceptionInformation())) {
21314|         Err = GetExceptionCode();
21315|         DLOG((TEXT("Exception %08x getting error
            | info\n"),Err));

```

```

21316| }
21317| return Err;
21318| }
21319|
21320| DLLEXPORT PSMSTATUS PSMAPI Psm_GetProgress( OUT
    | pPSM_GetProgressOut Out )
21321| {
21322|     ULONG Err=ERROR_INVALID_HANDLE;
21323|     pThreadStorage ThreadStorage = GetThreadStorage();
21324|
21325|     __try {
21326| //         DLOG((TEXT("Psm_GetProgress Called\n")));
21327|
21328|         if
            | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
            | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {
21329|             Err=PSMI_GetProgress(
                | ThreadStorage->PSManHandle, ThreadStorage->PSManEvent,
                | Out, NULL);
21330|         } else {
21331|             DLOG((TEXT("Error %08x psm not
                | installed\n"),Err));
21332|         }
21333|     }
        | __except(ExceptionFilter(GetExceptionInformation())) {
21334|         Err = GetExceptionCode();
21335|         DLOG((TEXT("Exception %08x getting
            | progress\n"),Err));
21336|     }
21337|     return Err;
21338| }
21339|
21340| DLLEXPORT PSMSTATUS PSMAPI Psm_GetSnapShotProgress( IN
    | pSnapShot SnapShot, OUT pPSM_GetProgressOut Out )
21341| {
21342|     ULONG Err=ERROR_INVALID_HANDLE;
21343|     pThreadStorage ThreadStorage = GetThreadStorage();
21344|
21345|     __try {
21346| //         DLOG((TEXT("Psm_GetProgress Called\n")));
21347|
21348|         if
            | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
            | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {
21349|             Err=PSMI_GetProgress(
                | ThreadStorage->PSManHandle, ThreadStorage->PSManEvent,
                | Out, SnapShot);
21350|         } else {
21351|             DLOG((TEXT("Error %08x psm not
                | installed\n"),Err));

```

```

21352|     }
21353| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
21354|     Err = GetExceptionCode();
21355|     DLOG((TEXT("Exception %08x getting
    | progress\n"),Err));
21356| }
21357| return Err;
21358| }
21359|
21360| DLLEXPORT PSMSTATUS PSMAPI Psm_OpenExclusive( pSnapShot
    | SnapShot )
21361| {
21362|     ULONG Err=ERROR_INVALID_HANDLE;
21363|     pThreadStorage ThreadStorage = GetThreadStorage();
21364|
21365|     __try {
21366|         DLOG((TEXT("Psm_OpenExclusive Called\n")));
21367|
21368|         if
            | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
            | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {
21369|             Err=PSMI_OpenExclusive(
            | ThreadStorage->PSManHandle, ThreadStorage->PSManEvent,
            | SnapShot );
21370|         } else {
21371|             DLOG((TEXT("Error %08x psm not
            | installed\n"),Err));
21372|         }
21373|     }
    | __except(ExceptionFilter(GetExceptionInformation())) {
21374|         Err = GetExceptionCode();
21375|         DLOG((TEXT("Exception %08x opening
            | exclusive\n"),Err));
21376|     }
21377|     return Err;
21378| }
21379|
21380| DLLEXPORT PSMSTATUS PSMAPI Psm_CloseExclusive(
    | pSnapShot SnapShot )
21381| {
21382|     ULONG Err=ERROR_INVALID_HANDLE;
21383|     pThreadStorage ThreadStorage = GetThreadStorage();
21384|
21385|     __try {
21386|         DLOG((TEXT("Psm_CloseExclusive Called\n")));
21387|
21388|         if
            | ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&
            | (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE)) {

```

```

21389|         Err=PSMI_CloseExclusive(
| ThreadStorage->PSManHandle, ThreadStorage->PSManEvent,
| SnapShot);
21390|     } else {
21391|         DLOG((TEXT("Error %08x psm not
| installed\n"),Err));
21392|     }
21393| }
| __except(ExceptionFilter(GetExceptionInformation())) {
21394|     Err = GetExceptionCode();
21395|     DLOG((TEXT("Exception %08x closing
| exclusive\n"),Err));
21396| }
21397| return Err;
21398| }
21399|
21400| DLLEXPORT PSMSTATUS PSMAPI Psm_CreateSnapShotW(
21401| IN pOpenTransactionIn3W In,
21402| IN ULONG NumVolumes,
21403| IN PVOID InVolumeMap,
21404| IN ULONG *VolumeMapFlags,
21405| IN ULONG OutVolumeMapByteSize,
21406| INOUT PVOID OutVolumeMap,
21407| OUT pOpenTransactionOutW Out,
21408| OUT pSnapShot *SnapShot,
21409| INOUT LPOVERLAPPED Overlapped,
21410| IN LPOVERLAPPED_COMPLETION_ROUTINE
| CompletionRoutine )
21411| {
21412|     ULONG Err=ERROR_INVALID_HANDLE;
21413|     pThreadStorage ThreadStorage = GetThreadStorage();
21414|     ULONG DoLogErr = 1;
21415|     ULONG DoCleanup = 1;
21416|
21417|     __try {
21418|         DLOG((TEXT("Psm_CreateSnapShot Called\n")));
21419|
21420|         if(ThreadStorage->RegisterCalled) {
21421|             if
| (ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) {
21422|                 if(!HasEvalExpired(NULL)) {
21423|                     if(Overlapped) {
21424|                         DLOG((TEXT("Creating PSM
| snapshot Async\n")));
21425|
21426|                         Err=PSMI_OpenEx(
| ThreadStorage->PSManHandle, In, NumVolumes,
| InVolumeMap, VolumeMapFlags, OutVolumeMapByteSize,
| OutVolumeMap, Out, Overlapped, CompletionRoutine,
| PSM_IFLAG_NEW_SNAPSHOT, SnapShot);

```

```

21427|          DLOG((TEXT("OpenEx returned
| %08x\n"),Err));
21428|          if(Err==ERROR_IO_PENDING) {
21429|              DoLogErr = 0;
21430|          }
21431|          DoCleanup=0;
21432|      } else {
21433|          OVERLAPPED o={0};
21434|
21435|          DLOG((TEXT("Creating PSM
| snapshot Sync\n"))));
21436|          | ResetEvent(ThreadStorage->PSManEvent);
21437|          o.hEvent =
| ThreadStorage->PSManEvent;
21438|
21439|          Err =
| PSMI_OpenEx(ThreadStorage->PSManHandle,ln,NumVolumes,
| InVolumeMap, VolumeMapFlags, OutVolumeMapByteSize,
| OutVolumeMap,
| Out,&o,NULL,PSM_IFLAG_NEW_SNAPSHOT,SnapShot);
21440|
21441|          if (Err == ERROR_IO_PENDING) {
21442|              DoLogErr = 0;
21443|              DoCleanup=0;
21444|              DLOG((TEXT("Waiting for
| Psm_Enable to finish\n"))));
21445|              // wait for app to finish
| running, or timeout
21446|              do {
21447|                  Err =
| WaitForSingleObjectEx ( ThreadStorage->PSManEvent,
| INFINITE, TRUE );
21448|                  DLOG((TEXT("WFSO
| finished %08x\n"),Err));
21449|              }
| while(Err==WAIT_IO_COMPLETION);
21450|
21451|          if(Err==WAIT_OBJECT_0)
| // get win32 error code
21452|              Err = o.InternalHigh;
| // o.Internal is the nt status code
21453|          else
21454|              Err = GetLastError();
21455|
21456|          DLOG((TEXT("Err=%08x ols:
| %08x %08x %08x %08x\n"), Err, o.InternalHigh,
| o.Internal, o.OffsetHigh, o.Offset,GetLastError()));
21457|      }
21458|  }

```

```

21459|         } else {
21460|             Err = PSM_EVALUATION_EXPIRED;
21461|             DLOG((TEXT("Error %08x psm eval
| expired\n"),Err));
21462|             PSMI_LogEvent( Err, Err, 0, NULL);
21463|             DoLogErr = 0;
21464|         }
21465|     } else {
21466|         DLOG((TEXT("Error %08x psm not
| installed\n"),Err));
21467|     }
21468| } else {
21469|     DLOG((TEXT("Psm_Register not
| called!\n"),Err));
21470|     Err = PSM_ERROR_REGISTER_NOT_CALLED;
21471|     PSMI_LogEvent( Err, Err, 0, NULL);
21472|     DoLogErr = 0;
21473| }
21474| }
| __except(ExceptionFilter(GetExceptionInformation())) {
21475|     Err = GetExceptionCode();
21476|     DLOG((TEXT("Exception %08x opening
| psm\n"),Err));
21477| }
21478| if(DoCleanup) {
21479|     // clean up message status the com object may
| have set
21480|     // as the kernel hasnt had a chance to execute
| and clean
21481|     // the message up.
21482|     Psm_SetStatus(PSM_IDLE,PSM_GLOBAL_STATUS);
21483| }
21484| if((DoLogErr) && (Err)) {
21485|     WCHAR *Strings[1];
21486|     WCHAR Error[20];
21487|     swprintf(Error,L"%08x",Err);
21488|     Strings[0] = Error;
21489|     PSMI_LogEvent(
| PSM_SNAPSHOT_COULD_NOT_BE_CREATED, Err, 1,Strings );
21490| }
21491| return Err;
21492| }
21493|
21494| DLLEXPORT PSMSTATUS PSMAPI Psm_CreateSnapShotA(
21495| IN pOpenTransactionIn3A In,
21496| IN ULONG NumVolumes,
21497| IN PVOID InVolumeMap,
21498| IN ULONG *VolumeMapFlags,
21499| IN ULONG OutVolumeMapByteSize,
21500| INOUT PVOID OutVolumeMap,

```



```

21501|  OUT pOpenTransactionOutA Out,
21502|  OUT pSnapshot *SnapShot,
21503|  INOUT LPOVERLAPPED Overlapped,
21504|  IN LPOVERLAPPED_COMPLETION_ROUTINE
    | CompletionRoutine )
21505| {
21506|  tOpenTransactionIn3W InW={0};
21507|  PWCHAR *Vmw=NULL;
21508|  ULONG Err=0;
21509|  ULONG i;
21510|  CHAR **InV;
21511|
21512|  __try {
21513|      if((!In) || (!Out) || (!InVolumeMap) ||
        | (!VolumeMapFlags) || (!OutVolumeMap) ||
        | (OutVolumeMapByteSize<8) || ((signed)NumVolumes<1)) {
21514|          try_return (Err=ERROR_INVALID_PARAMETER);
21515|      }
21516|
21517|      if(
21518|          (In->Size!=sizeof(tOpenTransactionIn3A)) &&
21519|          (In->Size!=sizeof(tOpenTransactionIn2A)) &&
21520|          (In->Size!=sizeof(tOpenTransactionIn1A))
21521|      ) {
21522|          try_return(Err=ERROR_INVALID_PARAMETER);
21523|      }
21524|
21525|      // check the length so we dont copy over the
        | stack and destory
21526|      // everything we are lookin at if the things
        | contain long names
21527|      if(
        | (!CheckForZeroTerminatorA(In->CacheFileName,256)) ||
21528|      | (!CheckForZeroTerminatorA(In->PreOpenFile,256)) ||
21529|      | (!CheckForZeroTerminatorA(In->PostOpenFile,256)) ||
21530|      | (!CheckForZeroTerminatorA(In->PostCloseFile,256)) ||
21531|          (!CheckForZeroTerminatorA(In->UserName,40))
        | ||
21532|          (!CheckForZeroTerminatorA(In->Password,40))
        | ) {
21533|          try_return(Err=ERROR_INVALID_PARAMETER);
21534|      }
21535|
21536|      InW.Size=sizeof(tOpenTransactionIn3W);
21537|      InW.SizeOfCacheFileMB = In->SizeOfCacheFileMB;
21538|      InW.MaxSizeOfCacheFileMB =
        | In->MaxSizeOfCacheFileMB;

```

```

21539|     InW.Flags = In->Flags;
21540|     InW.QuiescentWait = In->QuiescentWait;
21541|     InW.QuiescentTimeout = In->QuiescentTimeout;
21542|     InW.ErrorEvent = In->ErrorEvent;
21543|     InW.AbortEvent = In->AbortEvent;
21544|
21545|     // Persistent structure is same size as 3A so
    | above stuff copied all the right values
21546|     if(In->Size>=sizeof(tOpenTransactionIn2A)) {
21547|         InW.SecurityInfo = In->SecurityInfo;
21548|
21549|         if(In->Size>=sizeof(tOpenTransactionIn3A))
    | {
21550|             InW.CallerPrivateUse =
    | In->CallerPrivateUse;
21551|             InW.NumToKeep = In->NumToKeep;
21552|         } else {
21553|             InW.CallerPrivateUse = NULL;
21554|             InW.NumToKeep = -1;
21555|         }
21556|     } else {
21557|         InW.SecurityInfo = NULL;
21558|     }
21559|
21560|
    | OemToCharW(In->CacheFileName,InW.CacheFileName);
21561|     OemToCharW(In->PreOpenFile,InW.PreOpenFile);
21562|     OemToCharW(In->PostOpenFile,InW.PostOpenFile);
21563|
    | OemToCharW(In->PostCloseFile,InW.PostCloseFile);
21564|     OemToCharW(In->UserName,InW.UserName);
21565|     OemToCharW(In->Password,InW.Password);
21566|
21567|     (PVOID)InV = InVolumeMap;
21568|
21569|     Vmw =
    | LocalAlloc(LPTR,NumVolumes*sizeof(PVOID));
21570|     if(!Vmw) {
21571|         try_return(Err=ERROR_OUTOFMEMORY);
21572|     }
21573|
21574|     for(i=0;i<NumVolumes;i++) {
21575|         Vmw[i] =
    | LocalAlloc(LPTR,strlen(InV[i])*sizeof(WCHAR)+sizeof(WCHA
    | R));
21576|         if(!Vmw[i]) {
21577|             ULONG j;
21578|             for (j=0;j<i;j++) {
21579|                 LocalFree(Vmw[j]);
21580|             }

```

```

21581|         LocalFree(Vmw);
21582|         try_return(Err=ERROR_OUTOFMEMORY);
21583|     }
21584|     OemToCharW(InV[i],Vmw[i]);
21585| }
21586|
21587| // put special marker in out so the APC routine
| knows that
21588| // it should be ansi and not unicode
21589| (*(DWORD*)Out) = 0x5a4b3c2d;
21590|
21591| // In, InVolumeMap, InVolumeMapFlags are
| completely read only and not accessed after this
21592| // function returns.
21593| Err = Psm_CreateSnapShotW(&InW,NumVolumes, Vmw,
| VolumeMapFlags, OutVolumeMapByteSize, OutVolumeMap,
| (tOpenTransactionOutW*)Out,SnapShot,Overlapped,Completi
| nRoutine);
21594|
21595|     for(i=0;i<NumVolumes;i++) {
21596|         LocalFree(Vmw[i]);
21597|         Vmw[i] = NULL;
21598|     }
21599|     LocalFree(Vmw);
21600|
21601| try_exit:
21602|     // need to have a statement after a label
21603|     ;
21604| }
| __except(ExceptionFilter(GetExceptionInformation())) {
21605|     Err = GetExceptionCode();
21606|     DLOG((TEXT("Exception %08x converting ansi to
| wide during open\n"),Err));
21607| }
21608| return Err;
21609| }
21610|
21611|
21612| DLLEXPORT PSMSTATUS PSMAPI Psm_DestroySnapShot (
| tSnapShot *SnapShot )
21613| {
21614|     ULONG Err=ERROR_INVALID_HANDLE;
21615|     pThreadStorage ThreadStorage = GetThreadStorage();
21616|
21617|     __try {
21618|         DLOG((TEXT("Psm_DestroySnapShot Called\n")));
21619|         __try {
21620|             // close psm
21621|             if
| ((ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) &&

```

```

    | (ThreadStorage->PsmHandle!=INVALID_HANDLE_VALUE)) {
21622|         Err =
    | PSMI_Close(ThreadStorage->PsmHandle,ThreadStorage->PSM
    | anEvent,SnapShot);
21623|     }
21624| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
21625|     DLOG((TEXT("Exception %08x closing
    | psm\n"),Err));
21626|     Err = GetExceptionCode();
21627| }
21628| } __finally {
21629|     // make sure we decrement this so we can close
    | the handle
21630|     // and have the driver clean up after us, even
    | if the above call failed.
21631|     if(ThreadStorage->NumOpens>0)
21632|         ThreadStorage->NumOpens--;
21633|     if(ThreadStorage->NumOpens) {
21634|         DLOG((TEXT("Psm still open\n")));
21635|     }
21636| }
21637|
21638| return Err;
21639| }
21640|
21641| //-----
    | -----
21642| // Internal support functions
21643|
21644| #ifdef _DEBUG
21645| void PSM_LogDebugInfo( const TCHAR *fmt,...) {
21646|     FILE *fp=NULL;
21647|     va_list argptr;
21648|     TCHAR Buff[256];
21649|     TCHAR Buff2[256];
21650|     TCHAR Timestr[30];
21651|     time_t Time;
21652|
21653|     if(!DebugMode)
21654|         return;
21655|
21656|     va_start(argptr, fmt);
21657|     _vstprintf(Buff2, fmt, argptr);
21658|     va_end(argptr);
21659|
21660|     Time = time(NULL);
21661|     _tcsftime(Timestr,40,TEXT("%m/%d/%y %H:%M:%S:
    | "),localtime(&Time));
21662|

```

```

21663|   _sprintf(Buff,TEXT("%s%s"),Timestr,Buff2);
21664|
21665|   if(TlsIndex!=0xffffffff) {
21666|       pThreadStorage ThreadStorage =
           | GetThreadStorage();
21667|
21668|       | if(ThreadStorage->DebugLogFile!=INVALID_HANDLE_VALUE) {
21669|           ULONG Written;
21670|           // write to file..
21671|
           | WriteFile(ThreadStorage->DebugLogFile,Buff,_tcslen(Buff)
           | *sizeof(TCHAR),&Written,NULL);
21672|       }
21673|   }
21674|
21675|   // write to debugger...
21676|   OutputDebugString(Buff);
21677| }
21678| void PSM_LogDebugInfoW( const WCHAR *fmt,...) {
21679|   FILE *fp=NULL;
21680|   va_list argptr;
21681|   WCHAR Buff[256];
21682|   WCHAR Buff2[256];
21683|   WCHAR Timestr[30];
21684|   time_t Time;
21685|
21686|   if(!DebugMode)
21687|       return;
21688|
21689|   va_start(argptr, fmt);
21690|   vswprintf(Buff2, fmt, argptr);
21691|   va_end(argptr);
21692|
21693|   Time = time(NULL);
21694|   wcsftime(Timestr,40,L"%m/%d/%y %H:%M:%S:
           | ",localtime(&Time));
21695|
21696|   swprintf(Buff,L"%s%s",Timestr,Buff2);
21697|
21698|   if(TlsIndex!=0xffffffff) {
21699|       pThreadStorage ThreadStorage =
           | GetThreadStorage();
21700|
21701|       | if(ThreadStorage->DebugLogFile!=INVALID_HANDLE_VALUE) {
21702|           ULONG Written;
21703|           // write to file..
21704|
           | WriteFile(ThreadStorage->DebugLogFile,Buff,wcslen(Buff)*

```

```

    | sizeof(WCHAR),&Written,NULL);
21705|     }
21706| }
21707|
21708| // write to debugger...
21709| OutputDebugStringW(Buff);
21710| }
21711| #endif
21712|
21713| STATIC ULONG GetDeviceName (
21714|     HANDLE VolHandle,
21715|     WCHAR *DeviceName,
21716|     ULONG BufferLengthInChars )
21717| {
21718|     ULONG dwBytesReturned;
21719|     ULONG Err=ERROR_MORE_DATA;
21720|     BOOL B;
21721|     ULONG CurrentSize = 100; // sizeof volume in system
    | space, see mountmgr.h
21722|     PMOUNTDEV_NAME MN;
21723|
21724|     do {
21725|         MN =
    | LocalAlloc(LPTR,CurrentSize+sizeof(MOUNTDEV_NAME));
21726|         if(MN) {
21727|             if((B = DeviceIoControl(VolHandle,
21728|                 IOCTL_MOUNTDEV_QUERY_DEVICE_NAME,
21729|                 NULL, 0,
21730|                 MN, CurrentSize+sizeof(MOUNTDEV_NAME),
21731|                 &dwBytesReturned,
21732|                 NULL))) {
21733|                 Err = 0;
21734|             } else {
21735|                 Err = GetLastError();
21736|                 if(Err==ERROR_MORE_DATA) {
21737|                     LocalFree(MN);
21738|                     CurrentSize*=2;
21739|                 }
21740|             }
21741|         } else {
21742|             Err = ERROR_OUTOFMEMORY;
21743|             B = FALSE;
21744|         }
21745|     } while(Err==ERROR_MORE_DATA);
21746|
21747|     if(!Err) {
21748|         int numChars =
    | min((ULONG)MN->NameLength/sizeof(WCHAR),BufferLengthInCh
    | ars-1);
21749|         wcsncpy(DeviceName,MN->Name,numChars);

```

```

21750|    // null terminate
21751|    DeviceName[numChars] = 0;
21752| } else {
21753|     Err = GetLastError();
21754|     DLOG((TEXT("Error %08x getting device
    | name\n"),Err));
21755| }
21756| if(MN)
21757|     LocalFree(MN);
21758| return Err;
21759| }
21760|
21761| WCHAR NibbleToHexWChar( unsigned char In, BOOLEAN
    | TakeUpper )
21762| {
21763|     In = TakeUpper?(In >> 4):(In & 0xf);
21764|     return (WCHAR)( ( In > 9 )?(In - 10 + 'a'):(In +
    | '0') );
21765| }
21766|
21767| ULONG BufferToHexWChar( PVOID Buffer, ULONG NumBytes,
    | PWCHAR Out, ULONG *OutSize)
21768| {
21769|     ULONG Status = 0;
21770|     ULONG i;
21771|     unsigned char *In = (unsigned char*)Buffer;
21772|
21773|     //Round to take whole number of D_words
21774|     NumBytes = (NumBytes+3)/4*4;
21775|
21776|     if (Out==NULL) {
21777|         *OutSize = (NumBytes*2+1)*sizeof(WCHAR);
21778|     } else {
21779|         if (*OutSize < 2) {
21780|             // just give back bad status as not room to
    | put even an empty string
21781|             Status = ERROR_INSUFFICIENT_BUFFER;
21782|
21783|         } else {
21784|
21785|             if ((NumBytes*2+1)*sizeof(WCHAR) > *OutSize
    | ) {
21786|                 //not enough room for all we have pack
    | the limit and give bad status
21787|                 NumBytes =
    | (*OutSize/sizeof(WCHAR)-1)/2;
21788|                 Status = ERROR_BUFFER_OVERFLOW;
21789|             }
21790|             *OutSize = (NumBytes*2+1)*sizeof(WCHAR);
21791|

```

```

21792|         for (i=0;i<NumBytes;In++,i++,Out+=2) {
21793|             Out[0] = NibbleToHexWChar(In[0],1);
21794|             Out[1] = NibbleToHexWChar(In[0],0);
21795|         }
21796|         Out[0] = L'\0';
21797|     }
21798| }
21799| return Status;
21800| }
21801|
21802| ULONG GetUniqueldForVolume (
21803|     HANDLE VolHandle,
21804|     WCHAR *Uniqueld,
21805|     ULONG UniqueldLength
21806| )
21807| {
21808|     ULONG dwBytesReturned;
21809|     ULONG Err=0;
21810|     BOOL B;
21811|     BYTE UniqueRaw[16+sizeof(MOUNTDEV_UNIQUE_ID)];
21812|     PMOUNTDEV_UNIQUE_ID
        | UniquePtr=(PMOUNTDEV_UNIQUE_ID)UniqueRaw;
21813|
21814|     if(Uniqueld) {
21815|         if((B = DeviceIoControl(VolHandle,
21816|             IOCTL_MOUNTDEV_QUERY_UNIQUE_ID,
21817|             NULL, 0,
21818|             UniqueRaw, sizeof(UniqueRaw),
21819|             &dwBytesReturned,
21820|             NULL))) {
21821|             Err = BufferToHexWChar(
21822|                 UniquePtr->Uniqueld,
21823|                 | min(dwBytesReturned,UniquePtr->UniqueldLength),
21824|                 Uniqueld,
21825|                 &UniqueldLength);
21826|         } else {
21827|             Err = GetLastError();
21828|         }
21829|     }
21830|
21831|     return Err;
21832| }
21833|
21834| STATIC ULONG GetNTNameForUniqueld( WCHAR
        | *LookingForUniqueld, WCHAR *NTName, ULONG NTNameLen )
21835| {
21836|     ULONG Err = ERROR_INVALID_FUNCTION;
21837|     HANDLE Handle;
21838|     BOOL B;

```



```

21839|  WCHAR Buffer[256];
21840|  BOOL Leave=FALSE;
21841|
21842|  if((((GetVersion() & 0xffff0000) >> 16) > 1381) {
21843|      Handle = pFindFirstVolume( Buffer,
        | sizeof(Buffer)/sizeof(WCHAR) );
21844|      if(Handle!=INVALID_HANDLE_VALUE) {
21845|          do {
21846|              ULONG Good=FALSE;
21847|              UNICODE_STRING Uni;
21848|              HANDLE hVolume;
21849|              OBJECT_ATTRIBUTES
        | ObjectAttributes={0};
21850|              IO_STATUS_BLOCK IoStatus={0};
21851|              WCHAR UniqueId[20*4];
21852|              ULONG UniqueIdLength=sizeof(UniqueId);
21853|
21854|              switch(GetDriveTypeW(Buffer)) {
21855|                  case DRIVE_FIXED      :
21856|                      // if hard drive then good
21857|                      DLOG((TEXT("Drive '%S' is local
        | hard drive\n"),Buffer));
21858|                      Good=TRUE;
21859|                      break;
21860|                  case DRIVE_REMOVABLE  :
21861|                      DLOG((TEXT("Drive '%S' is
        | removable\n"),Buffer));
21862|                      Good=FALSE;
21863|                      break;
21864|                  case DRIVE_REMOTE     :
21865|                      DLOG((TEXT("Drive '%S' is not a
        | local hard drive\n"),Buffer));
21866|                      Good=FALSE;
21867|                      break;
21868|                  case DRIVE_CDROM      :
21869|                      DLOG((TEXT("Drive '%S' is not a
        | local hard drive\n"),Buffer));
21870|                      Good=FALSE;
21871|                      break;
21872|                  case DRIVE_RAMDISK    :
21873|                      DLOG((TEXT("Drive '%S' is not a
        | local hard drive\n"),Buffer));
21874|                      Good=FALSE;
21875|                      break;
21876|                  default:
21877|                      Good=FALSE;
21878|                      break;
21879|              }
21880|
21881|              if(Good) {

```

```

21882|          // Buffer = "\\?\Volume{xxx}\\"
21883|          Buffer[1] = L'?';
21884|          Buffer[wcslen(Buffer)-1]=0;
21885|          // Buffer = "\\??\Volume{xxx}"
21886|
21887|          RtlInitUnicodeString( &Uni,
| Buffer);
21888|
21889|          InitializeObjectAttributes (
| &ObjectAttributes,
21890|                      &Uni,
21891|                      | OBJ_CASE_INSENSITIVE,
21892|                      NULL,
21893|                      NULL
| );
21894|
21895|          Err = NtCreateFile( &hVolume,
21896|          | FILE_GENERIC_READ,          // desired access
21897|          | &ObjectAttributes,        // object attributes
21898|          | &IoStatus,
21899|          | NULL,
| // alloc size
21900|          | FILE_ATTRIBUTE_NORMAL,    // file attributes
21901|          | FILE_SHARE_WRITE | FILE_SHARE_READ,
| // share access
21902|          | FILE_OPEN,
| // create disposition
21903|          | FILE_SYNCHRONOUS_IO_NONALERT,          // create
| options
21904|          | NULL, //
| eabuffer
21905|          | 0 ); //
| ealength
21906|
21907|          if(!Err) {
21908|              Err =
| GetUniqueldForVolume(hVolume,Uniqueld,sizeof(Uniqueld));
21909|
21910|              if(!Err) {
21911|                  DLOG((TEXT("Volume '%S'
| unique id = '%S'\n"),Buffer,Uniqueld));
21912|
21913|                  if(!_wcsicmp(Uniqueld,LookingForUniqueld)==0 ) {

```

```

21914|             Err =
| GetDeviceName(hVolume,NTName,NTNameLen);
21915|             if(!Err) {
21916|             | DLOG((TEXT("Success! '%S' =
| '%S'\n"),LookingForUniqueId,NTName));
21917|             Leave = TRUE;
21918|             } else {
21919|             DLOG((TEXT("Error
| %08x getting device name\n"),Err));
21920|             }
21921|             }
21922|             } else {
21923|             DLOG((TEXT("Error %08x
| getting unique id\n"),Err));
21924|             }
21925|             NtClose(hVolume);
21926|             } else {
21927|             DLOG((TEXT("Error %08x opening
| volume\n"),Err));
21928|             }
21929|             } // if good
21930|
21931|             if(!Leave) {
21932|             B =
| pFindNextVolume(Handle,Buffer,sizeof(Buffer)/sizeof(WCHA
| R));
21933|             } else {
21934|             B = FALSE;
21935|             }
21936|             } while(B);
21937|             pFindVolumeClose(Handle);
21938|             } else {
21939|             Err = GetLastError();
21940|             DLOG((TEXT("Error %08x finding first
| volume\n"),Err));
21941|             }
21942|         }
21943|     return Err;
21944| }
21945|
21946| /* This procedure can convert the following passed in
| names
21947| 1. c:\
21948| 2. \DosDevices\C:
21949| 3. \DosDevices\HardDisk0\Partition1
21950| 4. \??\HardDisk0\Partition1
21951| 5. \?\Volume{e7df379c-5f0f-11d3-bb4e-806d6172696f}
21952| 6. \Device\HarddiskDmVolumes\W2kServer1Dg0\Volume1
21953| 7. \Device\HardDiskVolume1

```

```

21954| 8. c:\drivee\   where drivee is a junction point
21955| 9. 12345678000011112222 - this is the unique id
    | of the volume
21956| */
21957| STATIC ULONG GetNTDeviceName (
21958|     WCHAR *AName,
21959|     WCHAR *NTName,
21960|     ULONG BufferLengthInChars,
21961|     ULONG Fast )
21962| {
21963|     UNICODE_STRING Str;
21964|     ULONG Err=0;
21965|     WCHAR Temp[256]={0};
21966|     WCHAR *pName=AName;
21967|     int len = 0;
21968|
21969|     Str.Length=wcslen(AName)*sizeof(WCHAR);
21970|     Str.MaximumLength = Str.Length + sizeof(WCHAR);
21971|     Str.Buffer = AName;
21972|
21973|     // case 6 & 7
21974|     if(!_wcsnicmp(AName,L"\\Device\\",8)==0) {
21975|         // already a NT device name
21976|         wcsncpy(NTName,AName,BufferLengthInChars);
21977|         return 0;
21978|     }
21979|
21980|     // get rid of extra slash
21981|     if(Str.Buffer[(Str.Length/2)-1] == L'\\') {
21982|         Str.Length-=2;
21983|     }
21984|
21985|     if((((GetVersion() & 0xffff0000) >> 16) > 1381) &&
    | (!Fast)) {
21986| IsMountMgrName:
21987|         // case 5
21988|         if(MOUNTMGR_IS_VOLUME_NAME(&Str)) {
21989|             HANDLE hVolume = CreateFileW(
21990|                 pName,
21991|                 0,
21992|                 FILE_SHARE_READ | FILE_SHARE_WRITE,
21993|                 NULL,
21994|                 OPEN_EXISTING,
21995|                 0,
21996|                 NULL );
21997|             if(hVolume!=INVALID_HANDLE_VALUE) {
21998|                 Err =
    | GetDeviceName(hVolume,NTName,BufferLengthInChars);
21999|                 CloseHandle(hVolume);
22000|                 return Err;

```

```

22001|         } else {
22002|             Err = GetLastError();
22003|             DLOG((TEXT("Error %08x opening
| volume\n"),Err));
22004|             // fall through
22005|         }
22006|     }
22007|     // case 8
22008|
22009|     // will not be first time through
22010|     if(pName!=Temp) {
22011|         | if(pGetVolumeNameForVolumeMountPoint(AName,Temp,BufferLe
| ngthInChars)) {
22012|             pName = Temp;
22013|             Str.Length=wcslen(pName)*sizeof(WCHAR);
22014|             Str.MaximumLength = Str.Length +
| sizeof(WCHAR);
22015|             Str.Buffer = pName;
22016|             // get rid of extra slash
22017|             if(Str.Buffer[(Str.Length/2)-1] ==
| L'\\') {
22018|                 Str.Length-=2;
22019|                 Temp[(Str.Length/2)]=0;
22020|             }
22021|             DLOG((TEXT("GVNFVMP='%S'\n"),pName));
22022|             goto IsMountMgrName;
22023|         } else {
22024|             DLOG((TEXT("Error %08x getting volume
| for mount point\n"),GetLastError()));
22025|             // fall through
22026|         }
22027|     }
22028| } // of win2k specific stuff
22029|
22030| wscpy(Temp,AName);
22031| len = wcslen(Temp);
22032| if((len > 0) && (Temp[len-1] == L'\\')) {
22033|     Temp[len-1] = 0;
22034| }
22035|
22036| SetLastError(0);
22037| // case 1, 2, 3, 4
22038|
| if(QueryDosDeviceW(Temp,NTName,BufferLengthInChars)!=0)
| {
22039|     Err = 0;
22040| } else {
22041|     Err = GetLastError();
22042| }

```

```

22043|   if(Err) {
22044|       Err = GetNTNameForUniqueld( AName, NTName,
    | BufferLengthInChars );
22045|   }
22046|
22047|   DLOG((TEXT("GetNTDeviceName returning %08x: BL=%08x
    | '%S' = '%S'\n"),Err,BufferLengthInChars,Temp,NTName));
22048|   return Err;
22049| }
22050|
22051|
22052| STATIC ULONG MakeVolumeList (
22053|   ULONG NumVolumes,
22054|   pVolMap VolumeMap,
22055|   ULONG BufferLength,
22056|   pOpenTransactionInInternal Internal,
22057|   ULONG InternalBufferChars )
22058| {
22059|   ULONG i = 0;
22060|   ULONG Err = 0;
22061|   PCHAR Buffer = (PCHAR)Internal->DeviceName;
22062|   ULONG Len = 0;
22063|
22064|   // skip over list of pointers.
22065|   Buffer+=NumVolumes*sizeof(PVOID);
22066|   BufferLength-=NumVolumes*sizeof(PVOID);
22067|
22068|   for(i=0;i<NumVolumes;i++) {
22069|       if((Err =
    | GetNTDeviceName(VolumeMap[i],(WCHAR*)Buffer,InternalBuff
    | erChars,FALSE))!=0) {
22070|           return Err;
22071|       }
22072|       Internal->DeviceName[i] =
    | DN_MakeOffset(Internal,Buffer);
22073|
    | Len=wcslen((WCHAR*)Buffer)*sizeof(WCHAR)+sizeof(WCHAR);
22074|       Buffer+=Len;
22075|       BufferLength-=Len;
22076|       if((signed long)BufferLength<=0) {
22077|           DLOG((TEXT("Buffer too small\n")));
22078|           return ERROR_INSUFFICIENT_BUFFER;
22079|       }
22080|       DLOG((TEXT("MakeVolumeList: VolIndex=%lu, '%S'
    | =
    | '%S'\n"),i,VolumeMap[i],DN_MakePointer(Internal,Internal
    | ->DeviceName[i])));
22081|   }
22082|
22083|   Internal->NumberOfDevices = NumVolumes;

```

```

22084|    DLOG((TEXT("Number of volumes to psm =
      | %d\n"),NumVolumes));
22085|    return 0;
22086| }
22087|
22088| STATIC ULONG DriveLetterFree( WCHAR Drive )
22089| {
22090|     WCHAR DriveName[20];
22091|     WCHAR LinkName[256];
22092|
22093|     swprintf(DriveName,L"%c:",Drive);
22094|     // if not found then we can use it.
22095|     if(QueryDosDeviceW(DriveName,LinkName,256)==0)
22096|         return TRUE;
22097|     else
22098|         return FALSE;
22099| }
22100|
22101| STATIC WCHAR GetFirstFreeDriveLetter ( void )
22102| {
22103|     ULONG i;
22104|     WCHAR
      | DriveMap[]=L"0123456789~`!#$%*-_+[]{}'ABDEFGHIJKLMNOPQR
      | STUVWXYZ"; // C is skipped!!
22105|     ULONG Len=wcslen(DriveMap);
22106|
22107|     /*
22108|         Works = ~`!#$%*-_+[]{}'?.?
22109|         Doesnt = @^&(&|=|;|:|'|<|>|/
22110|         Iffy (ie it works but is it really good?) = .?
22111|     */
22112|
22113|     for(i=0;i<Len;i++) {
22114|         if (DriveLetterFree(DriveMap[i]))
22115|             return DriveMap[i];
22116|     }
22117|
22118|     // no drive letter found
22119|     return 0;
22120| }
22121|
22122| STATIC WCHAR GetFirstFreeNormalDriveLetter ( void )
22123| {
22124|     ULONG i;
22125|     WCHAR DriveMap[]=L"DEFGHIJKLMNOPQRSTUVWXYZAB"; // C
      | is skipped!!
22126|     ULONG Len=wcslen(DriveMap);
22127|
22128|     for(i=0;i<Len;i++) {
22129|         if (DriveLetterFree(DriveMap[i]))

```

```

22130|         return DriveMap[i];
22131|     }
22132|
22133|     // no drive letter found
22134|     return 0;
22135| }
22136|
22137| STATIC ULONG RemoveDeviceName ( WCHAR *NTDeviceName)
22138| {
22139|     ULONG GotitLocked=0;
22140|     ULONG Err=0;
22141|     WCHAR *ToUse=NULL;
22142|     UNICODE_STRING Uni={0};
22143|     OBJECT_ATTRIBUTES ObjectAttributes={0};
22144|     HANDLE hVolume;
22145|     IO_STATUS_BLOCK IoStatus={0};
22146|     ULONG Access;
22147|
22148|     DLOG((TEXT("RemoveDevice: Nt  =
| '%S'\n"),NTDeviceName));
22149|
22150|     RtlInitUnicodeString( &Uni, NTDeviceName);
22151|
22152|     InitializeObjectAttributes ( &ObjectAttributes,
22153|                                 &Uni,
22154|                                 OBJ_CASE_INSENSITIVE,
22155|                                 NULL,
22156|                                 NULL );
22157|
22158|     Access = FILE_GENERIC_READ | FILE_GENERIC_WRITE;
22159|
22160| DoOpen:
22161|     Err = NtCreateFile( &hVolume,
22162|                       Access,          //
| desired access
22163|                       &ObjectAttributes,
| // object attributes
22164|                       &IoStatus,
22165|                       NULL,           //
| alloc size
22166|                       FILE_ATTRIBUTE_NORMAL,
| // file attributes
22167|                       FILE_SHARE_WRITE |
| FILE_SHARE_READ,          // share
| access
22168|                       FILE_OPEN,      //
| create disposition
22169|                       | FILE_SYNCHRONOUS_IO_NONALERT,          // create
| options

```



```

22170|             NULL, // eabuffer
22171|             0 ); // ealength
22172|     if(Err) {
22173|         if(Access & FILE_WRITE_DATA) {
22174|             DLOG((TEXT("Error %08x opening '%S' for
| write %08x\n"),Err,NTDeviceName,Access));
22175|             Access = FILE_GENERIC_READ;
22176|         } else
22177|         if(Access & FILE_READ_DATA) {
22178|             DLOG((TEXT("Error %08x opening '%S' for
| read %08x\n"),Err,NTDeviceName,Access));
22179|             Access = 0;
22180|         } else {
22181|             DLOG((TEXT("Error %08x opening '%S' for
| query %08x\n"),Err,NTDeviceName,Access));
22182|             return Err;
22183|         }
22184|         goto DoOpen;
22185|     }
22186|
22187|     // Lock and dismount the volume.
22188|     // W2k doesnt need the lock
22189|     // this stuff superceded
22190| // if(((GetVersion() & 0xffff0000) >> 16) <= 1381) {
22191|
22192| //try and lockit so we can do an orderly dismount
22193| GotitLocked = FALSE; // LockVolume(hVolume);
22194|
22195|
22196| // under nt4 sp4 and greater you can dismount
| without locking first, so
22197| // try it anyway... the famous "forced dismount"
22198| if (DismountVolume(hVolume)) {
22199|
22200|     DLOG((TEXT("Dismounted '%S'\n"),NTDeviceName));
22201|
22202|     // Set prevent removal to false and eject the
| volume.
22203|     if (PreventRemovalOfVolume(hVolume, FALSE)) {
22204|         AutoEjectVolume(hVolume);
22205|     } else {
22206|         Err = GetLastError();
22207|     }
22208| } else {
22209|     Err = GetLastError();
22210|     DLOG((TEXT("Error %08x dismounting
| '%S'\n"),Err,NTDeviceName));
22211| }
22212|
22213| if (GotitLocked) {

```

```

22214|     UnlockVolume(hVolume);
22215| }
22216|
22217| // Close the volume so other processes can use the
    | drive.
22218| NtClose(hVolume);
22219|
22220| return Err;
22221| }
22222|
22223| STATIC ULONG AddWin32Link( WCHAR *Win32Link, WCHAR
    | *NTDeviceName )
22224| {
22225| #if 1
22226|     tPSM_SetWin32Link Link;
22227|     ULONG returned;
22228|     ULONG B=FALSE;
22229|     pThreadStorage ThreadStorage = GetThreadStorage();
22230|
22231|     wcscpy(Link.Win32Link,L"\\?\\");
22232|     wcscat(Link.Win32Link,Win32Link);
22233|     wcscpy(Link.NTDeviceName,NTDeviceName);
22234|
22235|     Link.Operation = LINK_SetLink;
22236|
22237|     DLOG(("Linking '%S' to
    | '%S\\n",Link.Win32Link,Link.NTDeviceName));
22238|     B = DeviceIoControl( ThreadStorage->PSManHandle,
22239|         IOCTL_SET_WIN32_LINK,
22240|         &Link,
22241|         sizeof(Link),
22242|         NULL,
22243|         0,
22244|         &returned,
22245|         NULL);
22246|     return B ? 0 : GetLastError();
22247| #else
22248|     ULONG B = DefineDosDeviceW( DDD_RAW_TARGET_PATH,
    | Win32Link, NTDeviceName);
22249|     if(B) {
22250|         return 0;
22251|     } else {
22252|         return GetLastError();
22253|     }
22254| #endif
22255| }
22256|
22257|
22258| STATIC ULONG RemoveWin32Link( WCHAR *Win32Link )
22259| {

```

```

22260| #if 1
22261|     tPSM_SetWin32Link Link;
22262|     ULONG B=FALSE;
22263|     ULONG returned;
22264|     pThreadStorage ThreadStorage = GetThreadStorage();
22265|
22266|     wcscpy(Link.Win32Link,L"\\??\\");
22267|     wcscat(Link.Win32Link,Win32Link);
22268|     Link.Operation = LINK_DeleteLink;
22269|
22270|     DLOG(("Deleting Linking '%S'\n",Link.Win32Link));
22271|     B = DeviceIoControl( ThreadStorage->PSManHandle,
22272|         IOCTL_SET_WIN32_LINK,
22273|         &Link,
22274|         sizeof(Link),
22275|         NULL,
22276|         0,
22277|         &returned,
22278|         NULL);
22279|     return B ? 0 : GetLastError();
22280| #else
22281|     ULONG B = DefineDosDeviceW(DDD_RAW_TARGET_PATH |
        | DDD_REMOVE_DEFINITION,Win32Link, NTDeviceName);
22282|     if(B) {
22283|         return 0;
22284|     } else {
22285|         return GetLastError();
22286|     }
22287| #endif
22288| }
22289|
22290|
22291| STATIC ULONG RemoveDevice ( tMapDrive *PsmObject )
22292| {
22293|     ULONG Err=0;
22294|
22295|     DLOG((TEXT("RemoveDevice: Nt =
        | '%S'\n"),PsmObject->NTDeviceName));
22296|     DLOG((TEXT("RemoveDevice: Dos =
        | '%S'\n"),PsmObject->DriveLetterName));
22297|     DLOG((TEXT("RemoveDevice: Share=
        | '%S'\n"),PsmObject->ShareName));
22298|     DLOG((TEXT("RemoveDevice: Win32=
        | '%S'\n"),PsmObject->VolumeName));
22299|
22300|     // remove win32 crap
22301|     if(wcscmp(PsmObject->ShareName,L"")!=0) {
22302|         Err = NetShareDel( NULL, PsmObject->ShareName,
        | 0 );
22303|     }

```

```

22304|
22305|     if(wcscmp(PsmObject->DriveLetterName,L"")!=0) {
22306| //         if(DefineDosDeviceW(DDD_RAW_TARGET_PATH |
        | DDD_REMOVE_DEFINITION,PsmObject->DriveLetterName,
        | PsmObject->NTDeviceName)) {
22307|         Err =
        | RemoveWin32Link(PsmObject->DriveLetterName);
22308|     }
22309|
22310|     if(wcscmp(PsmObject->VolumeName,L"")!=0) {
22311| //         if(DefineDosDeviceW(DDD_RAW_TARGET_PATH |
        | DDD_REMOVE_DEFINITION,PsmObject->VolumeName,
        | PsmObject->NTDeviceName)) {
22312|         Err = RemoveWin32Link(PsmObject->VolumeName);
22313|     }
22314|
22315|     Err = RemoveDeviceName(PsmObject->NTDeviceName);
22316|     return Err;
22317| }
22318|
22319|
22320| STATIC WCHAR *RemoveInvalidChars( WCHAR *Name )
22321| {
22322|     ULONG Len=wcslen(Name);
22323|     WCHAR *p=Name;
22324|
22325|     while(*p!=L'\0') {
22326|         | if(wcschr(L"~!@#$$%^&*()-=\\+|[]{}<>,.?/\"";*p)!=NULL
        | ) {
22327|             // found invalid char, remove it
22328|             memmove(p,p+1,Len-(p-Name));
22329|             Name[Len] = 0;
22330|             Len--;
22331|         } else {
22332|             p++;
22333|         }
22334|     }
22335|     return Name;
22336| }
22337|
22338| // for mount of volumes
22339| STATIC ULONG TouchVolumeName( WCHAR *NTDeviceName )
22340| {
22341|     ULONG Err=0;
22342|     ULONG OldMode;
22343|     UNICODE_STRING Uni={0};
22344|     OBJECT_ATTRIBUTES ObjectAttributes={0};
22345|     HANDLE hVolume;
22346|     IO_STATUS_BLOCK IoStatus={0};

```

[illegible]

```

22388|     if(Access & FILE_WRITE_DATA) {
22389|         DLOG((TEXT("Error %08x Touching '%S' for
| write %08x\n"),Err,Name,Access));
22390|         Access = FILE_GENERIC_READ;
22391|     } else
22392|         if(Access & FILE_READ_DATA) {
22393|             DLOG((TEXT("Error %08x Touching '%S' for
| read %08x\n"),Err,Name,Access));
22394|             Access = 0;
22395|         } else {
22396|             DLOG((TEXT("Error %08x Touching '%S' for
| query %08x\n"),Err,Name,Access));
22397|             Err = GetLastError();
22398|             goto Done;
22399|         }
22400|         goto DoOpen;
22401|     }
22402| Done:
22403|     NtClose(hVolume);
22404|     LocalFree(Name);
22405|
22406|     SetErrorMode(OldMode);
22407|     return Err;
22408| }
22409|
22410| // for mount of volumes
22411| STATIC ULONG TouchVolume( tMapDrive *PsmObject )
22412| {
22413|     return TouchVolumeName(PsmObject->NTDeviceName);
22414| }
22415|
22416| ULONG GetWin32NameForNtDeviceName( WCHAR *NTName, WCHAR
| *Win32Name )
22417| {
22418|     WCHAR *Buffer,*p;
22419|     WCHAR Name[256];
22420|
22421|     Buffer =
| (WCHAR*)LocalAlloc(LPTR,65536*sizeof(WCHAR));
22422|     QueryDosDeviceW(NULL,Buffer,65536);
22423|
22424|     p = Buffer;
22425|     while(*p) {
22426|         QueryDosDeviceW(p,Name,256);
22427|         if(_wcsicmp(NTName,Name)==0) {
22428|             wcsncpy(Win32Name,p);
22429|         }
22430|         p = p + wcslen(p)+1;
22431|     }
22432|

```

```

22433| LocalFree(Buffer);
22434| return 0;
22435| }
22436|
22437| ULONG GetVolumeGuidForNtDeviceName( WCHAR *NTName,
    | WCHAR *VolumeGuid)
22438| {
22439|     WCHAR *Buffer,*p;
22440|     WCHAR Name[256];
22441|
22442|     wcscpy(VolumeGuid,L "");
22443|
22444|     Buffer =
    | (WCHAR*)LocalAlloc(LPTR,65536*sizeof(WCHAR));
22445|     QueryDosDeviceW(NULL,Buffer,65536);
22446|
22447|     p = Buffer;
22448|     while(*p) {
22449|         QueryDosDeviceW(p,Name,256);
22450|         if(_wcsicmp(NTName,Name)==0) {
22451|             // Volume{}
22452|             if((p[0]==L'V') && (p[1]==L'o') &&
    | (p[6]==L'{') {
22453|                 wcscpy(VolumeGuid,p);
22454|                 return 0;
22455|             }
22456|         }
22457|         p = p + wcslen(p)+1;
22458|     }
22459|
22460|     LocalFree(Buffer);
22461|     return ERROR_NOT_FOUND;
22462| }
22463|
22464|
22465| ULONG GetDriveLetterForNtDeviceName( WCHAR *NTName,
    | WCHAR *Win32Name )
22466| {
22467|     WCHAR *Buffer,*p;
22468|     WCHAR Name[256];
22469|     ULONG FoundOne=FALSE;
22470|
22471|     Buffer =
    | (WCHAR*)LocalAlloc(LPTR,65536*sizeof(WCHAR));
22472|     QueryDosDeviceW(NULL,Buffer,65536);
22473|
22474|     p = Buffer;
22475|     while(*p) {
22476|         QueryDosDeviceW(p,Name,256);
22477|         if( (_wcsicmp(NTName,Name)==0) &&

```

```

22478|      (*(p+1)==L':') {
22479|          wcsncpy(Win32Name,p);
22480|          FoundOne = TRUE;
22481|          break;
22482|      }
22483|      p = p + wcslen(p)+1;
22484|  }
22485|
22486|  LocalFree(Buffer);
22487|  if(FoundOne) {
22488|      return 0;
22489|  } else {
22490|      return ERROR_FILE_NOT_FOUND;
22491|  }
22492| }
22493|
22494|
22495| /*
22496|  Returns back device names to application that they
22497|  | can use
22498|  FIXFIXFIX need to return back a user accessible
22499|  | address, not the NT
22500|  device name
22501|  */
22502| /*
22503|  STATIC ULONG AddDrivesToSystem( tAPCContext *ApcContext
22504|  | )
22505|  {
22506|      ULONG BS = sizeof(PVOID) *
22507|      | ApcContext->OTO->NumberOfDevices;
22508|      ULONG i = 0;
22509|      PCHAR Buffer = ((char*)ApcContext->VolumeMap)+BS;
22510|      ULONG Len = 0;
22511|      pVolMap Map = 0;
22512|      tMapDrive *PsmObject = 0;
22513|      ULONG Status = 0;
22514|      WCHAR *ToUse = 0;
22515|      ULONG StartAdding = 0;
22516|      pThreadStorage ThreadStorage = GetThreadStorage();
22517|
22518|      if ( BS > ApcContext->SizeOfVolumeMap ) {
22519|          return ERROR_INSUFFICIENT_BUFFER;
22520|      }
22521|
22522|      WaitForSingleObject ( SharedMemoryMutex, INFINITE
22523|      | );
22524|      __try {
22525|          StartAdding =
22526|          | SharedMemory->NumberOfMappedVolumes;
22527|
22528|

```



```

22522|     Map = ApcContext->VolumeMap;
22523|
22524|     for(i=0;i<ApcContext->OTI->NumberOfDevices;i++)
        | {
22525|         DLOG(("Mapping in %08x:%08x:'%S' =
        | '%S'\n",ApcContext->VolumeMapFlags[i],ApcContext->OTI->D
        | eviceName[i],DN_MakePointer(ApcContext->OTI,ApcContext->
        | OTI->DeviceName[i]),DN_MakePointer(ApcContext->OTO,ApcCo
        | ntext->OTO->DeviceName[i]));
22526|
22527|         PsmObject =
        | &SharedMemory->MappedVolumes[SharedMemory->NumberOfMappe
        | dVolumes];
22528|         memset(PsmObject,0,sizeof(tMapDrive));
22529|
22530|         | wcscpy(PsmObject->NTDeviceName,DN_MakePointer(ApcContext
        | ->OTO,ApcContext->OTO->DeviceName[i]));
22531|         | wcscpy(PsmObject->OriginalNTName,DN_MakePointer(ApcConte
        | xt->OTI,ApcContext->OTI->DeviceName[i]));
22532|         | wcscpy(PsmObject->Company,ThreadStorage->Company);
22533|         | wcscpy(PsmObject->Product,ThreadStorage->Product);
22534|         | wcscpy(PsmObject->Version,ThreadStorage->Version);
22535|         | wcscpy(PsmObject->Code,ThreadStorage->Code);
22536|         wcscpy(PsmObject->Key,ThreadStorage->Key);
22537|
22538|         | GetWin32NameForNtDeviceName(PsmObject->OriginalNTName,Ps
        | mObject->OriginalUserName);
22539|         | if(wcscmp(PsmObject->OriginalUserName,L"")==0) {
22540|         | wcscpy(PsmObject->OriginalUserName,PsmObject->OriginalNT
        | Name);
22541|         }
22542|         ToUse = NULL;
22543|
22544|         // force volume to be mounted
22545|         TouchVolume(PsmObject);
22546|
22547|         if(ApcContext->VolumeMapFlags[i] &
        | PSM_VOLUME_MAP_VOLUME) {
22548|             // make a win32 device object
22549|             | swprintf(PsmObject->VolumeName,L"Psm%s",&PsmObject->NTDe

```

```

    | viceName[24]);
22550|          // NT name =
    | '\Device\PsmDevices_0110\ _Device_HarddiskDmVolumes_W2kse
    | rver1Dg0_Volume1_0'
22551|          // skip over '\Device\PsmDevices_0110\'
22552|          DLOG(("Mapping '%S' to
    | '%S'\n",PsmObject->VolumeName,
    | PsmObject->NTDeviceName));
22553| //          if(!DefineDosDeviceW(
    | DDD_RAW_TARGET_PATH, PsmObject->VolumeName,
    | PsmObject->NTDeviceName)) {
22554|          if((Status = AddWin32Link(
    | PsmObject->VolumeName, PsmObject->NTDeviceName))) {
22555|          DLOG((TEXT("Error %08x mapping
    | win32 name\n"),Status));
22556|          try_return(Status = Status);
22557|          }
22558|          ToUse = PsmObject->VolumeName;
22559|          }
22560|          if((ApcContext->VolumeMapFlags[i] &
    | PSM_VOLUME_MAP_DRIVE_LETTER)) {
22561|          WCHAR DriveLetter;
22562|
22563|          if((ApcContext->VolumeMapFlags[i] &
    | PSM_VOLUME_MAP_SHARE) || (ApcContext->VolumeMapFlags[i]
    | & PSM_VOLUME_MAP_USE_LETTER)) {
22564|
    | DriveLetter=GetFirstFreeNormalDriveLetter();
22565|          } else {
22566|
    | DriveLetter=GetFirstFreeDriveLetter();
22567|          }
22568|
22569|          // map drive letter
22570|          if(DriveLetter!=0) {
22571|
    | swprintf(PsmObject->DriveLetterName,L"%c:",DriveLetter);
22572|
22573|          DLOG(("Mapping '%S' to
    | '%S'\n",PsmObject->DriveLetterName,
    | PsmObject->NTDeviceName));
22574|          //if(!DefineDosDeviceW(
    | DDD_RAW_TARGET_PATH, PsmObject->DriveLetterName,
    | PsmObject->NTDeviceName)) {
22575|          if((Status = AddWin32Link(
    | PsmObject->DriveLetterName, PsmObject->NTDeviceName)))
    | {
22576|          DLOG((TEXT("Error %08x mapping
    | drive\n"),Status));
22577|          try_return(Status = Status);

```

```

22578|         }
22579|         ToUse = PsmObject->DriveLetterName;
22580|     } else {
22581|         DLOG((TEXT("Error! out of drive
| letters\n")));
22582|         try_return(Status =
| ERROR_CANCELLED);
22583|     }
22584| }
22585| if(ApcContext->VolumeMapFlags[i] &
| PSM_VOLUME_MAP_SHARE) {
22586|     // make a network share
22587|     // can only share if there is a "dos"
| device
22588|     if(ToUse) {
22589|         WCHAR *Temp =
| LocalAlloc(LPTR,wcslen(ToUse)*sizeof(WCHAR)+2*sizeof(WCH
| AR));
22590|         if(Temp) {
22591|             swprintf(Temp,L"%s\\",ToUse);
22592|             | swprintf(PsmObject->ShareName,L"Psm_ %s",ToUse);
22593|             | RemoveInvalidChars(PsmObject->ShareName);
22594|             DLOG(("Sharing '%S' as
| '%S'\n",Temp, PsmObject->ShareName));
22595|             Status = VDisk_MakeShareEx2(
| PsmObject->ShareName, Temp, &ApcContext->SecurityInfo);
22596|             LocalFree(Temp);
22597|             if(Status) {
22598|                 DLOG((TEXT("Error %08x
| sharing\n"),Status));
22599|                 | wcsncpy(PsmObject->ShareName,L "");
22600|                 goto try_exit;
22601|             }
22602|         } else {
22603|             DLOG(("Error out of
| memory\n"));
22604|         }
22605|     }
22606| }
22607|
22608| if(!ToUse)
22609|     ToUse = PsmObject->NTDeviceName;
22610|
22611| if((* (DWORD*)ApcContext->Out) ==
| 0x5a4b3c2d) {
22612|     // ansi version
22613|

```

```

    | Len=(wcslen(ToUse)*sizeof(CHAR))+sizeof(CHAR);
22614|     } else {
22615|         // unicode
22616|
    | Len=(wcslen(ToUse)*sizeof(WCHAR))+sizeof(WCHAR);
22617|     }
22618|     BS+=Len;
22619|     if(BS>ApcContext->SizeOfVolumeMap) {
22620|         try_return(Status =
    | ERROR_INSUFFICIENT_BUFFER);
22621|     }
22622|
22623|     Map[i]=(PWCHAR)Buffer;
22624|     if((* (DWORD*)ApcContext->Out) ==
    | 0x5a4b3c2d) {
22625|         CharToOemW(ToUse,Buffer);
22626|     } else {
22627|         wcscpy((PWCHAR)Buffer,ToUse);
22628|     }
22629|     Buffer+=Len;
22630|
22631|     PsmObject->SnapShotOffset    =
    | MakeOffset(*ApcContext->SnapShot);
22632|     SharedMemory->NumberOfMappedVolumes++;
22633|     Status = 0;
22634| }
22635| try_exit:
22636|     // remove all drives added.
22637|     if(Status) {
22638|
    | for(i=SharedMemory->NumberOfMappedVolumes;i>StartAdding;
    | i--) {
22639|
    | RemoveDevice(&SharedMemory->MappedVolumes[i]);
22640|     }
22641|     SharedMemory->NumberOfMappedVolumes =
    | StartAdding;
22642|     }
22643| } __finally {
22644|     ReleaseMutex(SharedMemoryMutex);
22645| }
22646|
22647| return Status;
22648| }
22649| */
22650|
22651| // {6B23C937-106E-4f94-967D-E65B7E5FAD09}
22652| #include <initguid.h>
22653| DEFINE_GUID(PSM_VDISK_GUID, 0x6b23c937, 0x106e, 0x4f94,
    | 0x96, 0x7d, 0xe6, 0x5b, 0x7e, 0x5f, 0xad, 0x9);

```

```

22654|
22655| ULONG CreateRootSnapShotDirectory ( WCHAR *Root )
22656| {
22657|     DWORD dwRes;
22658|     ULONG Err;
22659|     PSID pEveryoneSID = NULL, pAdminSID = NULL,
        | pSystemSID = NULL;
22660|     PACL pACL = NULL;
22661|     PSECURITY_DESCRIPTOR pSD = NULL;
22662|     EXPLICIT_ACCESS ea[3];
22663|     SID_IDENTIFIER_AUTHORITY SIDAAuthWorld =
        | SECURITY_WORLD_SID_AUTHORITY;
22664|     SID_IDENTIFIER_AUTHORITY SIDAAuthLocal =
        | SECURITY_LOCAL_SID_AUTHORITY;
22665|     SID_IDENTIFIER_AUTHORITY SIDAAuthNT =
        | SECURITY_NT_AUTHORITY;
22666|     SECURITY_ATTRIBUTES sa;
22667|
22668|     // Create a SID for the BUILTIN\Administrators
        | group.
22669|
22670|     if(! AllocateAndInitializeSid( &SIDAuthNT, 2,
22671|                                     SECURITY_BUILTIN_DOMAIN_RID,
22672|                                     DOMAIN_ALIAS_RID_ADMINS,
22673|                                     0, 0, 0, 0, 0, 0,
22674|                                     &pAdminSID) ) {
22675|         printf( "AllocateAndInitializeSid Error %u\n",
        | GetLastError() );
22676|         goto Cleanup;
22677|     }
22678|
22679|     // Create a well-known SID for the Everyone group.
22680|
22681|     if(! AllocateAndInitializeSid( &SIDAuthWorld, 1,
22682|                                     SECURITY_WORLD_RID,
22683|                                     0, 0, 0, 0, 0, 0, 0,
22684|                                     &pEveryoneSID) ) {
22685|         printf( "AllocateAndInitializeSid Error %u\n",
        | GetLastError() );
22686|         goto Cleanup;
22687|     }
22688|
22689|     // Create a well-known SID for the system group.
22690|
22691|     if(! AllocateAndInitializeSid( &SIDAuthNT, 1,
22692|                                     SECURITY_LOCAL_SYSTEM_RID,
22693|                                     0, 0, 0, 0, 0, 0, 0,
22694|                                     &pSystemSID) ) {
22695|         printf( "AllocateAndInitializeSid Error %u\n",
        | GetLastError() );

```

```

22696|     goto Cleanup;
22697| }
22698|
22699| // Initialize an EXPLICIT_ACCESS structure for an
    | ACE.
22700| // The ACE will allow the Administrators group full
    | access to the key.
22701|
22702| ZeroMemory(&ea, 3 * sizeof(EXPLICIT_ACCESS));
22703| ea[0].grfAccessPermissions =
22704|     GENERIC_READ |
22705|     GENERIC_ALL |
22706|     GENERIC_EXECUTE |
22707|     GENERIC_WRITE |
22708|     SPECIFIC_RIGHTS_ALL |
22709|     STANDARD_RIGHTS_ALL |
22710|     SYNCHRONIZE;
22711| ;
22712| ea[0].grfAccessMode = SET_ACCESS;
22713| ea[0].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
22714| ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
22715| ea[0].Trustee.TrusteeType = TRUSTEE_IS_GROUP;
22716| ea[0].Trustee.ptstrName = (LPTSTR) pAdminSID;
22717|
22718| ea[1].grfAccessPermissions =
22719|     GENERIC_READ |
22720|     GENERIC_ALL |
22721|     GENERIC_EXECUTE |
22722|     GENERIC_WRITE |
22723|     SPECIFIC_RIGHTS_ALL |
22724|     STANDARD_RIGHTS_ALL |
22725|     SYNCHRONIZE;
22726|
22727| ea[1].grfAccessMode = SET_ACCESS;
22728| ea[1].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
22729| ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
22730| ea[1].Trustee.TrusteeType = TRUSTEE_IS_USER;
22731| ea[1].Trustee.ptstrName = (LPTSTR) pSystemSID;
22732|
22733| ea[2].grfAccessPermissions = GENERIC_READ;
22734| ea[2].grfAccessMode = SET_ACCESS;
22735| ea[2].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
22736| ea[2].Trustee.TrusteeForm = TRUSTEE_IS_SID;
22737| ea[2].Trustee.TrusteeType =
    | TRUSTEE_IS_WELL_KNOWN_GROUP;
22738| ea[2].Trustee.ptstrName = (LPTSTR) pEveryoneSID;
22739|

```

```

22740|
22741| // Create a new ACL that contains the new ACEs.
22742|
22743| #define ONLY_ADMIN          1
22744| #define ADMIN_AND_SYSTEM    2
22745| #define ADMIN_SYSTEM_AND_EVERYONE 3
22746|
22747| dwRes = SetEntriesInAcl(ADMIN_AND_SYSTEM, ea, NULL,
    | &pACL);
22748| if (ERROR_SUCCESS != dwRes) {
22749|     SetLastError(dwRes);
22750|     printf( "SetEntriesInAcl Error %u\n",
    | GetLastError() );
22751|     goto Cleanup;
22752| }
22753|
22754| // Initialize a security descriptor.
22755|
22756| pSD = (PSECURITY_DESCRIPTOR) LocalAlloc(LPTR,
    | SECURITY_DESCRIPTOR_MIN_LENGTH);
22757| if (pSD == NULL) {
22758|     printf( "LocalAlloc Error %u\n", GetLastError()
    | );
22759|     goto Cleanup;
22760| }
22761|
22762| if (!InitializeSecurityDescriptor(pSD,
    | SECURITY_DESCRIPTOR_REVISION)) {
22763|     printf( "InitializeSecurityDescriptor Error
    | %u\n",
22764|           GetLastError() );
22765|     goto Cleanup;
22766| }
22767|
22768| // Add the ACL to the security descriptor.
22769|
22770| if (!SetSecurityDescriptorDacl(pSD,
22771|     TRUE,    // fDaclPresent flag
22772|     pACL,
22773|     FALSE)) // not a default DACL
22774| {
22775|     printf( "SetSecurityDescriptorDacl Error %u\n",
    | GetLastError() );
22776|     goto Cleanup;
22777| }
22778|
22779| // Initialize a security attributes structure.
22780|
22781| sa.nLength = sizeof (SECURITY_ATTRIBUTES);
22782| sa.lpSecurityDescriptor = pSD;

```

```

22783|  sa.bInheritHandle = FALSE;
22784|
22785|  // Use the security attributes to set the security
    | descriptor
22786|  // when you create a key.
22787|
22788|  if(CreateDirectoryW(Root,&sa)) {
22789|      // new directory
22790|  #if 0
22791|      // set if snapshot directory must be hidden
22792|
    | if(!SetFileAttributesW(Root,FILE_ATTRIBUTE_HIDDEN)) {
22793|      printf("Error %08x setting
    | attributes\n",GetLastError());
22794|      goto Cleanup;
22795|  }
22796| #endif
22797|  }
22798|
22799|  SetLastError(0);
22800| Cleanup:
22801|
22802|  Err = GetLastError();
22803|
22804|  if (pEveryoneSID) {
22805|      FreeSid(pEveryoneSID);
22806|  }
22807|  if (pSystemSID) {
22808|      FreeSid(pSystemSID);
22809|  }
22810|  if (pAdminSID) {
22811|      FreeSid(pAdminSID);
22812|  }
22813|  if (pACL) {
22814|      LocalFree(pACL);
22815|  }
22816|  if (pSD) {
22817|      LocalFree(pSD);
22818|  }
22819|  return Err;
22820| }
22821|
22822| ULONG CreatePSMWorkDirectory ( WCHAR *Root )
22823| {
22824|  DWORD dwRes;
22825|  ULONG Err;
22826|  PSID pEveryoneSID = NULL, pAdminSID = NULL,
    | pSystemSID = NULL;
22827|  PACL pACL = NULL;
22828|  PSECURITY_DESCRIPTOR pSD = NULL;

```



```

22829|  EXPLICIT_ACCESS ea[3];
22830|  SID_IDENTIFIER_AUTHORITY SIDAAuthWorld =
    | SECURITY_WORLD_SID_AUTHORITY;
22831|  SID_IDENTIFIER_AUTHORITY SIDAAuthLocal =
    | SECURITY_LOCAL_SID_AUTHORITY;
22832|  SID_IDENTIFIER_AUTHORITY SIDAAuthNT =
    | SECURITY_NT_AUTHORITY;
22833|  SECURITY_ATTRIBUTES sa;
22834|
22835|  // Create a SID for the BUILTIN\Administrators
    | group.
22836|
22837|  if(! AllocateAndInitializeSid( &SIDAuthNT, 2,
22838|                                SECURITY_BUILTIN_DOMAIN_RID,
22839|                                DOMAIN_ALIAS_RID_ADMINS,
22840|                                0, 0, 0, 0, 0, 0,
22841|                                &pAdminSID) ) {
22842|      printf( "AllocateAndInitializeSid Error %u\n",
    | GetLastError() );
22843|      goto Cleanup;
22844|  }
22845|
22846|  // Create a well-known SID for the Everyone group.
22847|
22848|  if(! AllocateAndInitializeSid( &SIDAuthWorld, 1,
22849|                                SECURITY_WORLD_RID,
22850|                                0, 0, 0, 0, 0, 0, 0,
22851|                                &pEveryoneSID) ) {
22852|      printf( "AllocateAndInitializeSid Error %u\n",
    | GetLastError() );
22853|      goto Cleanup;
22854|  }
22855|
22856|  // Create a well-known SID for the system group.
22857|
22858|  if(! AllocateAndInitializeSid( &SIDAuthNT, 1,
22859|                                SECURITY_LOCAL_SYSTEM_RID,
22860|                                0, 0, 0, 0, 0, 0, 0,
22861|                                &pSystemSID) ) {
22862|      printf( "AllocateAndInitializeSid Error %u\n",
    | GetLastError() );
22863|      goto Cleanup;
22864|  }
22865|
22866|  // Initialize an EXPLICIT_ACCESS structure for an
    | ACE.
22867|  // The ACE will allow the Administrators group full
    | access to the key.
22868|
22869|  ZeroMemory(&ea, 3 * sizeof(EXPLICIT_ACCESS));

```

```

22870|  ea[0].grfAccessPermissions =
22871|          STANDARD_RIGHTS_ALL |
22872|          STANDARD_RIGHTS_REQUIRED |
22873|          SYNCHRONIZE |
22874|          FILE_ADD_FILE | // this is so
    | we can create files in the directory
22875|          FILE_READ_ATTRIBUTES |
22876|          FILE_WRITE_ATTRIBUTES;
22877|
22878| ;
22879|  ea[0].grfAccessMode = SET_ACCESS;
22880|  ea[0].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
22881|  ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
22882|  ea[0].Trustee.TrusteeType = TRUSTEE_IS_GROUP;
22883|  ea[0].Trustee.ptstrName = (LPTSTR) pAdminSID;
22884|
22885|  ea[1].grfAccessPermissions =
22886|          GENERIC_READ |
22887|          GENERIC_ALL |
22888|          GENERIC_EXECUTE |
22889|          GENERIC_WRITE |
22890|          SPECIFIC_RIGHTS_ALL |
22891|          STANDARD_RIGHTS_ALL |
22892|          SYNCHRONIZE;
22893|
22894|  ea[1].grfAccessMode = SET_ACCESS;
22895|  ea[1].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
22896|  ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
22897|  ea[1].Trustee.TrusteeType = TRUSTEE_IS_USER;
22898|  ea[1].Trustee.ptstrName = (LPTSTR) pSystemSID;
22899|
22900|  ea[2].grfAccessPermissions = GENERIC_READ;
22901|  ea[2].grfAccessMode = SET_ACCESS;
22902|  ea[2].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
22903|  ea[2].Trustee.TrusteeForm = TRUSTEE_IS_SID;
22904|  ea[2].Trustee.TrusteeType =
    | TRUSTEE_IS_WELL_KNOWN_GROUP;
22905|  ea[2].Trustee.ptstrName = (LPTSTR) pEveryoneSID;
22906|
22907|
22908|  // Create a new ACL that contains the new ACEs.
22909|
22910| #define ONLY_ADMIN          1
22911| #define ADMIN_AND_SYSTEM    2
22912| #define ADMIN_SYSTEM_AND_EVERYONE  3
22913|
22914|  dwRes = SetEntriesInAcl(ADMIN_AND_SYSTEM, ea, NULL,

```

```

    | &pACL);
22915|   if (ERROR_SUCCESS != dwRes) {
22916|       SetLastError(dwRes);
22917|       printf( "SetEntriesInAcl Error %u\n",
    | GetLastError() );
22918|       goto Cleanup;
22919|   }
22920|
22921|   // Initialize a security descriptor.
22922|
22923|   pSD = (PSECURITY_DESCRIPTOR) LocalAlloc(LPTR,
    | SECURITY_DESCRIPTOR_MIN_LENGTH);
22924|   if (pSD == NULL) {
22925|       printf( "LocalAlloc Error %u\n", GetLastError()
    | );
22926|       goto Cleanup;
22927|   }
22928|
22929|   if (!InitializeSecurityDescriptor(pSD,
    | SECURITY_DESCRIPTOR_REVISION)) {
22930|       printf( "InitializeSecurityDescriptor Error
    | %u\n",
22931|           GetLastError() );
22932|       goto Cleanup;
22933|   }
22934|
22935|   // Add the ACL to the security descriptor.
22936|
22937|   if (!SetSecurityDescriptorDacl(pSD,
22938|       TRUE,    // fDaclPresent flag
22939|       pACL,
22940|       FALSE)) // not a default DACL
22941|   {
22942|       printf( "SetSecurityDescriptorDacl Error %u\n",
    | GetLastError() );
22943|       goto Cleanup;
22944|   }
22945|
22946|   // Initialize a security attributes structure.
22947|
22948|   sa.nLength = sizeof (SECURITY_ATTRIBUTES);
22949|   sa.lpSecurityDescriptor = pSD;
22950|   sa.bInheritHandle = FALSE;
22951|
22952|   // Use the security attributes to set the security
    | descriptor
22953|   // when you create a key.
22954|
22955|   if(CreateDirectoryW(Root,&sa)) {
22956|       // new directory

```

```

22957| #if 0
22958|     // set if snapshot directory must be hidden
22959|
22960|     | if(!SetFileAttributesW(Root,FILE_ATTRIBUTE_HIDDEN)) {
22961|         printf("Error %08x setting
22962|         | attributes\n",GetLastError());
22963|         goto Cleanup;
22964|     }
22965| #endif
22966| }
22967| SetLastError(0);
22968| Cleanup:
22969| Err = GetLastError();
22970|
22971| if (pEveryoneSID) {
22972|     FreeSid(pEveryoneSID);
22973| }
22974| if (pSystemSID) {
22975|     FreeSid(pSystemSID);
22976| }
22977| if (pAdminSID) {
22978|     FreeSid(pAdminSID);
22979| }
22980| if (pACL) {
22981|     LocalFree(pACL);
22982| }
22983| if (pSD) {
22984|     LocalFree(pSD);
22985| }
22986| return Err;
22987| }
22988|
22989|
22990| STATIC ULONG AddDrivesToSystemPersistent( tAPCContext
22991| | *ApcContext, PLARGE_INTEGER Time )
22992| {
22993|     ULONG
22994|     | BS=sizeof(PVOID)*ApcContext->OTO->NumberOfDevices;
22995|     ULONG i;
22996|     PCHAR Buffer=((char*)ApcContext->VolumeMap)+BS;
22997|     ULONG Len;
22998|     ULONG Status = 0;
22999|     WCHAR *ToUse=NULL;
23000|     pVolMap Map;
23001|     WCHAR SnapShotDirName[256];
23002|     if(BS>ApcContext->SizeOfVolumeMap) {
23003|         return ERROR_INSUFFICIENT_BUFFER;

```

```

23003|    }
23004|
23005|    Map = ApcContext->VolumeMap;
23006|
23007|    MakeSnapShotDirectoryName(SnapShotDirName,
    | ApcContext->In.CacheFileName,ApcContext->OTO->KernelSnap
    | ShotPointer);
23008|
23009|    // check for duplicate names on all volumes
    | that this snapshot has in it
23010|    {
23011|        WCHAR Dir[1024];
23012|        ULONG Index=0;
23013|
23014|        i=0;
23015|        wcsncpy(Dir,SnapShotDirName);
23016|        DLOG(("AddDrivesToSystemPersistent:
    | NumberOfDevices=%d\n",ApcContext->OTI->NumberOfDevices))
    | ;
23017|
23018|        while(i<ApcContext->OTI->NumberOfDevices) {
23019|            WCHAR VolumeName[100];
23020|            HANDLE DirHandle;
23021|            WIN32_FIND_DATAW FindFileData;
23022|            ULONG NoInc = FALSE;
23023|
23024|            if(GetDriveLetterForNtDeviceName(DN_MakePointer(ApcConte
    | xt->OTI,ApcContext->OTI->DeviceName[i]),
    | VolumeName)!=0) {
23025|                DLOG(("AddDrivesToSystemPersistent:
    | (1) DeviceName[i] =
    | '%S'\n",i,DN_MakePointer(ApcContext->OTI,ApcContext->OTI
    | ->DeviceName[i] ));
23026|                GetWin32NameForNtDeviceName(
    | DN_MakePointer(ApcContext->OTI,ApcContext->OTI->DeviceNa
    | me[i]), VolumeName);
23027|                DLOG(("AddDrivesToSystemPersistent:
    | (2) DeviceName[i] =
    | '%S'\n",i,DN_MakePointer(ApcContext->OTI,ApcContext->OTI
    | ->DeviceName[i] ));
23028|            }
23029|
23030|            // at this point we want
    | "C:\\snapshots\\hourly"
23031|            // VolumeName contains something like
    | "C:" or "\\device\\HardDiskVolume1"
23032|            // SnapShotDirName contains something
    | like 'snapshots\\hourly' or 'snapshots\\my snapshot at
    | 5:00pm'

```

```

23033|         if(Index==0) {
23034|
23035|             | swprintf(Dir,L"%s\\%s",VolumeName,SnapShotDirName);
23036|         } else {
23037|             | swprintf(Dir,L"%s\\%s.%d",VolumeName,SnapShotDirName,Ind
23038|             | ex);
23039|         }
23040|         // See if directory already exists
23041|         DirHandle = FindFirstFileW( Dir,
23042|             | &FindFileData);
23043|         while(DirHandle!=INVALID_HANDLE_VALUE)
23044|         | {
23045|             // shoot, the directory exists, so
23046|             | lets try and make it unique
23047|             FindClose(DirHandle);
23048|             Index++;
23049|             | swprintf(Dir,L"%s\\%s.%d",VolumeName,SnapShotDirName,Ind
23050|             | ex);
23051|             // start back at beginning looking
23052|             | for dups
23053|             NoInc = TRUE;
23054|             i=0;
23055|             DirHandle = FindFirstFileW( Dir,
23056|             | &FindFileData);
23057|             } // while
23058|             if(NoInc) {
23059|                 NoInc=FALSE;
23060|             } else {
23061|                 i++;
23062|             }
23063|             } // while
23064|             if(Index>0) {
23065|                 | swprintf(SnapShotDirName,L"%s.%d",SnapShotDirName,Index)
23066|                 | ;
23067|             }
23068|             }
23069|             // make the name persist
23070|             | Psm_SetUserName(ApcContext->OTO->KernelSnapShotPointer,S

```

```

    | napShotDirName,sizeof(SnapShotDirName));
23070|
23071|     for(i=0;i<ApcContext->OTI->NumberOfDevices;i++)
    | {
23072|         WCHAR Dir[1024];
23073|         WCHAR Link[256];
23074|         WCHAR BigBuffer[1024];
23075|
23076|         ToUse = NULL;
23077|
23078|         DLOG(("Mapping in %08x:%08x:'%S' = '%S'\n",
23079|             ApcContext->VolumeMapFlags[i],
23080|             ApcContext->OTI->DeviceName[i],
23081|
    | DN_MakePointer(ApcContext->OTI,ApcContext->OTI->DeviceNa
    | me[i]),
23082|
    | DN_MakePointer(ApcContext->OTO,ApcContext->OTO->DeviceNa
    | me[i])));
23083|
23084|         if(ApcContext->VolumeMapFlags[i] &
    | PSM_VOLUME_MAP_MOUNT_POINT) {
23085|
23086|
    | if(GetDriveLetterForNtDeviceName(DN_MakePointer(ApcConte
    | xt->OTI,ApcContext->OTI->DeviceName[i]), BigBuffer)!=0)
    | {
23087|             GetWin32NameForNtDeviceName(
    | DN_MakePointer(ApcContext->OTI,ApcContext->OTI->DeviceNa
    | me[i]), BigBuffer);
23088|         }
23089|
23090|
    | swprintf(Dir,L"%s\\%s",BigBuffer,SnapShotLocation);
23091|         // make root dir for snapshots on this
    | volume if it
23092|         // doesnt exist
23093|         CreateRootSnapShotDirectory(Dir);
23094|
23095|
    | swprintf(Dir,L"%s\\%s",BigBuffer,SnapShotDirName);
23096|         // remove the dir if it exists ( we
    | just proved it didnt in the above code.. rob
    | 12-19-2000) as
23097|         // set file time will not work if the
    | directory is a junction point
23098|         RemoveDirectoryW(Dir);
23099|         // Create and set the time of the
    | directory
23100|         SetTimeForFile( Dir, Time );

```

```

23101|
23102|     | wcsncpy(Link,DN_MakePointer(ApcContext->OTO,ApcContext->O
      | TO->DeviceName[i]));
23103|
23104|         ToUse = Dir;
23105|
23106|         Status =
      | CreateJunction2(Dir,Link,Time);
23107|         if(Status==0) {
23108|             // force volume to be mounted
23109|             TouchVolumeName(Link);
23110|         } else {
23111|             if((ApcContext->VolumeMapFlags[i] &
      | PSM_VOLUME_MAP_DRIVE_LETTER)) {
23112|                 // lets try and map a drive
      | letter, as it may be fat
23113|                 ToUse = NULL;
23114|             } else {
23115|                 ToUse = Link;
23116|                 // make the name persist
23117|
      | Psm_SetUserName(ApcContext->OTO->KernelSnapShotPointer,T
      | oUse,sizeof(ToUse));
23118|         }
23119|     }
23120| }
23121|     if((ApcContext->VolumeMapFlags[i] &
      | PSM_VOLUME_MAP_DRIVE_LETTER)) {
23122|         WCHAR DriveLetter;
23123|
23124|         if(!ToUse) {
23125|             // if no mount points set, then
      | lets do a drive letter
23126|
23127|             if((ApcContext->VolumeMapFlags[i] &
      | PSM_VOLUME_MAP_SHARE) || (ApcContext->VolumeMapFlags[i]
      | & PSM_VOLUME_MAP_USE_LETTER)) {
23128|
      | DriveLetter=GetFirstFreeNormalDriveLetter();
23129|             } else {
23130|
      | DriveLetter=GetFirstFreeDriveLetter();
23131|             }
23132|
23133|             // map drive letter
23134|             if(DriveLetter!=0) {
23135|
      | swprintf(Link,L"%c:",DriveLetter);
23136|

```



```

23137|                DLOG(("Mapping '%S' to
| '%S'\n",Link,
| DN_MakePointer(ApcContext->OTO,ApcContext->OTO->DeviceNa
| me[i])););
23138|                //if(DefineDosDeviceW(
| DDD_RAW_TARGET_PATH, Link,
| DN_MakePointer(ApcContext->OTO,ApcContext->OTO->DeviceNa
| me[i])) {
23139|                Status = AddWin32Link( Link,
| DN_MakePointer(ApcContext->OTO,ApcContext->OTO->DeviceNa
| me[i]));
23140|                if(Status==0) {
23141|                    ToUse = Link;
23142|                    // make the name persist
23143|                    // lets not do this for
| drive letters
23144| //
| Psm_SetUserName(ApcContext->OTO->KernelSnapShotPointer,T
| oUse,sizeof(ToUse));
23145|                } else {
23146|                    DLOG((TEXT("Error %08x
| mapping drive\n"),Status));
23147|                }
23148|                } else {
23149|                    DLOG((TEXT("Error! out of drive
| letters\n"))););
23150|                Status = ERROR_CANCELLED;
23151|                }
23152|                }
23153|                }
23154|                if(ApcContext->VolumeMapFlags[i] &
| PSM_VOLUME_MAP_SHARE) {
23155|                    // share the volume FIXFIXFIX
23156|                }
23157|                if(ApcContext->VolumeMapFlags[i] &
| PSM_VOLUME_MAP_VOLUME) {
23158|                    if(!ToUse) {
23159|                        // make a win32 volume guid.
23160|                        GetWin32NameForNtDeviceName(
| DN_MakePointer(ApcContext->OTI,ApcContext->OTI->DeviceNa
| me[i]), BigBuffer);
23161|                        ToUse = BigBuffer;
23162|                        // make the name persist
23163|
| Psm_SetUserName(ApcContext->OTO->KernelSnapShotPointer,T
| oUse,sizeof(ToUse));
23164|                }
23165|                }
23166|
23167|                if(!ToUse) {

```

```

23168|          // use the nt device name
23169|          ToUse =
        | DN_MakePointer(ApcContext->OTI,ApcContext->OTI->DeviceNa
        | me[i]);
23170|          // make the name persist
23171|          | Psm_SetUserName(ApcContext->OTO->KernelSnapShotPointer,T
        | oUse,sizeof(ToUse));
23172|          }
23173|
23174|          // -----
23175|          if((* (DWORD*)ApcContext->Out) ==
        | 0x5a4b3c2d) {
23176|              // ansi version
23177|              | Len=(wcslen(ToUse)*sizeof(CHAR))+sizeof(CHAR);
23178|          } else {
23179|              // unicode
23180|              | Len=(wcslen(ToUse)*sizeof(WCHAR))+sizeof(WCHAR);
23181|          }
23182|          BS+=Len;
23183|          if(BS>ApcContext->SizeOfVolumeMap) {
23184|              Status = ERROR_INSUFFICIENT_BUFFER;
23185|              return Status;
23186|          }
23187|
23188|          Map[i]=(PWCHAR)Buffer;
23189|          if((* (DWORD*)ApcContext->Out) ==
        | 0x5a4b3c2d) {
23190|              CharToOemW(ToUse,Buffer);
23191|          } else {
23192|              wcscpy((PWCHAR)Buffer,ToUse);
23193|          }
23194|          Buffer+=Len;
23195|
23196|          Status = 0;
23197|      }
23198|      return Status;
23199| }
23200|
23201|
23202| // SharedMemoryMutex MUST be acquired before calling
        | this routine
23203| STATIC ULONG FreeVolumesForSnapShot ( pSnapShot
        | SnapShot )
23204| {
23205|     ULONG Err=0;
23206|     ULONG i=0;
23207|     while(i<SharedMemory->NumberOfMappedVolumes) {

```

```

23208|     DLOG((TEXT("RVFS: %d: %08x (%08x) %08x
| (%08x)\n"),i,SharedMemory->MappedVolumes[i].SnapShotOffs
| et,MakePointer(SharedMemory->MappedVolumes[i].SnapShotOf
| fset),MakeOffset(SnapShot),SnapShot));
23209|
23210|
| if(MakePointer(SharedMemory->MappedVolumes[i].SnapShotOf
| fset) == SnapShot) {
23211|
| RemoveDevice(&SharedMemory->MappedVolumes[i]);
23212|
| memmove(&SharedMemory->MappedVolumes[i],&SharedMemory->M
| appedVolumes[i+1],(SharedMemory->NumberOfMappedVolumes-i
| -1)*sizeof(SharedMemory->MappedVolumes[0]));
23213|     SharedMemory->NumberOfMappedVolumes--;
23214|     // dont inc i since we moved everything
| down one
23215| } else {
23216|     i++;
23217| }
23218| }
23219| return Err;
23220| }
23221|
23222| STATIC ULONG FreeOneVolumeForSnapShot ( pSnapShot
| SnapShot, WCHAR *NTName )
23223| {
23224|     ULONG Err=0;
23225|     ULONG i=0;
23226|
23227|     DLOG(("FreeOneVolumeForSnapShot\n"));
23228|
23229|     WaitForSingleObject ( SharedMemoryMutex, INFINITE
| );
23230|     __try {
23231|
23232|         while(i<SharedMemory->NumberOfMappedVolumes) {
23233|             DLOG((TEXT("RVFS: %d: %08x (%08x) %08x
| (%08x)\n"),i,SharedMemory->MappedVolumes[i].SnapShotOffs
| et,MakePointer(SharedMemory->MappedVolumes[i].SnapShotOf
| fset),MakeOffset(SnapShot),SnapShot));
23234|
23235|
| if((MakePointer(SharedMemory->MappedVolumes[i].SnapShotO
| ffset) == SnapShot) &&
23236|
| ((_wcsicmp(SharedMemory->MappedVolumes[i].OriginalNTName
| ,NTName)==0) ||
23237|
| (_wcsicmp(SharedMemory->MappedVolumes[i].NTDeviceName,NT

```

```

    | Name)==0))) {
23238|
    | RemoveDevice(&SharedMemory->MappedVolumes[i]);
23239|
    | memmove(&SharedMemory->MappedVolumes[i],&SharedMemory->M
    | appedVolumes[i+1],(SharedMemory->NumberOfMappedVolumes-i
    | -1)*sizeof(SharedMemory->MappedVolumes[0]));
23240|         SharedMemory->NumberOfMappedVolumes--;
23241|         // dont inc i since we moved everything
    | down one
23242|         } else {
23243|             i++;
23244|         }
23245|     }
23246| } __finally {
23247|     ReleaseMutex(SharedMemoryMutex);
23248| }
23249| return Err;
23250| }
23251|
23252| STATIC ULONG GetOriginalNameFromVDiskName( WCHAR
    | *VDiskName, WCHAR *NTName )
23253| {
23254|     ULONG Err=0;
23255|     ULONG i=0;
23256|
23257|     WaitForSingleObject ( SharedMemoryMutex, INFINITE
    | );
23258|     __try {
23259|
23260|         while(i<SharedMemory->NumberOfMappedVolumes) {
23261|
    | if(!_wcsicmp(SharedMemory->MappedVolumes[i].NTDeviceName,
    | NTName)==0) {
23262|
    | wcsncpy(NTName,SharedMemory->MappedVolumes[i].OriginalINTN
    | ame);
23263|             break;
23264|         } else {
23265|             i++;
23266|         }
23267|     }
23268| } __finally {
23269|     ReleaseMutex(SharedMemoryMutex);
23270| }
23271| return Err;
23272| }
23273|
23274| STATIC ULONG GetVirtualFromOriginal(pSnapShot
    | SnapShot,WCHAR *OriginalDrive, WCHAR *VirtualDrive,

```

```

    | ULONG Size)
23275| {
23276|     ULONG Err=0;
23277|     ULONG i=0;
23278|
23279|     WaitForSingleObject ( SharedMemoryMutex, INFINITE
    | );
23280|     __try {
23281|
23282|         while(i<SharedMemory->NumberOfMappedVolumes) {
23283|
    | if((_wcsicmp(SharedMemory->MappedVolumes[i].OriginalNTNa
    | me,OriginalDrive)==0) &&
23284|
    | (SharedMemory->MappedVolumes[i].SnapShotOffset ==
    | MakeOffset(SnapShot))) {
23285|
    | wcscpy(VirtualDrive,SharedMemory->MappedVolumes[i].NTDev
    | iceName);
23286|         break;
23287|     } else {
23288|         i++;
23289|     }
23290| }
23291| } __finally {
23292|     ReleaseMutex(SharedMemoryMutex);
23293| }
23294| return Err;
23295| }
23296|
23297|
23298| STATIC ULONG RemoveAllVDisksForThread( )
23299| {
23300|     ULONG i;
23301|     ULONG Err=0;
23302|     pThreadStorage ThreadStorage = GetThreadStorage();
23303|
23304|     DLOG(("RemoveAllVDisksForThread\n"));
23305|
23306|     WaitForSingleObject ( SharedMemoryMutex, INFINITE
    | );
23307|     __try {
23308|         for(i=ThreadStorage->NumSnapShots;i>0;i--) {
23309|
    | FreeVolumesForSnapShot(MakePointer(ThreadStorage->SnapSh
    | otsOffset[i-1]));
23310|         }
23311|         ThreadStorage->NumSnapShots = 0;
23312|
23313|     } __finally {

```

```

23314|     ReleaseMutex(SharedMemoryMutex);
23315| }
23316| return Err;
23317|
23318| }
23319|
23320| STATIC ULONG PSMI_IsAnPSMVolume( WCHAR *VolumeName )
23321| {
23322|     ULONG Err=ERROR_INVALID_PARAMETER;
23323|     ULONG i;
23324|
23325|     WaitForSingleObject ( SharedMemoryMutex, INFINITE
        | );
23326|
23327|     __try {
23328|
        | for(i=0;i<SharedMemory->NumberOfMappedVolumes;i++) {
23329|         if(
        | (_wcsicmp(SharedMemory->MappedVolumes[i].DriveLetterName
        | ,VolumeName)==0) ||
23330|         | (_wcsicmp(SharedMemory->MappedVolumes[i].VolumeName,Volu
        | meName)==0) ||
23331|         | (_wcsicmp(SharedMemory->MappedVolumes[i].ShareName,Volum
        | eName)==0) ||
23332|         | (_wcsicmp(SharedMemory->MappedVolumes[i].NTDeviceName,Vo
        | lumeName)==0)) {
23333|             Err = 0;
23334|             break;
23335|         } else {
23336|             Err = ERROR_INVALID_PARAMETER;
23337|         }
23338|     }
23339| } __finally {
23340|     ReleaseMutex(SharedMemoryMutex);
23341| }
23342| return Err==0 ? TRUE : FALSE;
23343| }
23344|
23345| #define MOUNTMGR_IS_VOLUME_NAME_STRING(s) (
        | \
23346|     (wcslen(s) == 96 || (wcslen(s) == 98 && (s)[48] ==
        | L'\\')) && \
23347|     (s)[0] == L'\\' &&
        | \
23348|     ((s)[1] == L'?' || (s)[1] == L'\\') &&
        | \
23349|     (s)[2] == L'?' &&

```

```

| \
23350| (s)[3] == L'\\' &&
| \
23351| (s)[4] == L'V' &&
| \
23352| (s)[5] == L'o' &&
| \
23353| (s)[6] == L'l' &&
| \
23354| (s)[7] == L'u' &&
| \
23355| (s)[8] == L'm' &&
| \
23356| (s)[9] == L'e' &&
| \
23357| (s)[10] == L'{' &&
| \
23358| (s)[19] == L'-' &&
| \
23359| (s)[24] == L'-' &&
| \
23360| (s)[29] == L'-' &&
| \
23361| (s)[34] == L'-' &&
| \
23362| (s)[47] == L'}'
| \
23363| )
23364|
23365| STATIC ULONG PSMI_CanBePSMed( WCHAR *VolumeName, ULONG
| *VolumeType )
23366| {
23367|     WCHAR Buffer[256];
23368|     UNICODE_STRING Str;
23369|     HANDLE hVolume;
23370|     ULONG Err;
23371|
23372|     *VolumeType = PSM_IS_UNKNOWN;
23373|
23374|     Err = GetNTNameForUniqueId( VolumeName, Buffer,
| 256);
23375|     if(Err) {
23376|         // check to see if valid win32/NT name
23377|         if((Err =
| GetNTDeviceName(VolumeName,(WCHAR*)Buffer,256,FALSE))!=0
| ) {
23378|             DLOG((TEXT("Error %08x getting device
| name\n"),Err));
23379|             return FALSE;
23380|         }

```

```

23381| } else {
23382|     // unique id
23383|     *VolumeType = PSM_IS_SUPPORTED;
23384|     return TRUE;
23385| }
23386|
23387| if(PSMI_IsAnPSMVolume(Buffer)) {
23388|     // can not psm an psm volume
23389|     DLOG((TEXT("'%S' is PSM volume
    | '%S'\n"),VolumeName,Buffer));
23390|     *VolumeType = PSM_IS_PSM;
23391|     return FALSE;
23392| }
23393|
23394| // if it like so \??\G:\ then it is a subst drive
23395| if( (Buffer[0]==L'\\') &&
23396|     (Buffer[1]==L'?) &&
23397|     (Buffer[2]==L'?) &&
23398|     (Buffer[3]==L'\\')) {
23399|
23400|     if(!MOUNTMGR_IS_VOLUME_NAME_STRING(Buffer)) {
23401|         DLOG((TEXT("'%S' is a symbolic link to
    | '%S'\n"),VolumeName,Buffer));
23402|         *VolumeType = PSM_IS_SUBST;
23403|         return FALSE;
23404|     }
23405| }
23406|
23407| // make "C:" into "C:\
23408| wcsncpy(Buffer,VolumeName);
23409| if(Buffer[wcslen(Buffer)-1] != L'\\')
23410|     wcscat(Buffer,L"\\");
23411|
23412| switch(GetDriveTypeW(Buffer)) {
23413|     case DRIVE_REMOVABLE :
23414|     case DRIVE_FIXED :
23415|         // if hard drive then good
23416|         DLOG((TEXT("Drive '%S' is local hard
    | drive\n"),Buffer));
23417|         *VolumeType = PSM_IS_SUPPORTED;
23418|         return TRUE;
23419|     case DRIVE_REMOTE :
23420|         *VolumeType = PSM_IS_REMOTE;
23421|         DLOG((TEXT("Drive '%S' is not a local hard
    | drive\n"),Buffer));
23422|         return FALSE;
23423|     case DRIVE_CDROM :
23424|         *VolumeType = PSM_IS_CDROM;
23425|         DLOG((TEXT("Drive '%S' is not a local hard
    | drive\n"),Buffer));

```



```

23426|         return FALSE;
23427|     case DRIVE_RAMDISK    :
23428|         *VolumeType = PSM_IS_VIRTUAL;
23429|         DLOG((TEXT("Drive '%S' is not a local hard
| drive\n"),Buffer));
23430|         return FALSE;
23431|     case 1:
23432|     default:
23433|         DLOG((TEXT("Do not know what Drive '%S' is
| yet\n"),Buffer));
23434|         break;
23435|     } // switch
23436|
23437|     // Hmm, if Win2k check to see if volume name
23438|
23439|     if((((GetVersion() & 0xffff0000) >> 16) > 1381) {
23440|         DLOG((TEXT("Doing win2k checks\n")));
23441|
23442|         Str.Length=wcslen(VolumeName)*sizeof(WCHAR);
23443|         Str.MaximumLength = Str.Length + sizeof(WCHAR);
23444|         Str.Buffer = VolumeName;
23445|
23446|     IsMountMgrName:
23447|         // if mount manager name
23448|         if(MOUNTMGR_IS_VOLUME_NAME(&Str)) {
23449|             DLOG((TEXT("Volume is mount manager
| volume\n")));
23450|             *VolumeType = PSM_IS_SUPPORTED;
23451|             return TRUE;
23452|         }
23453|
23454|         if(Str.Buffer == VolumeName) {
23455|             // if it is a mount point and a local
| volume
23456|
| if(pGetVolumeNameForVolumeMountPoint(VolumeName,Buffer,2
| 56)) {
23457|
23458|             DLOG((TEXT("Volume is mount
| point\n")));
23459|
| Str.Length=wcslen(Buffer)*sizeof(WCHAR);
23460|             Str.MaximumLength = Str.Length +
| sizeof(WCHAR);
23461|             Str.Buffer = Buffer;
23462|             // get rid of extra slash
23463|             if(Str.Buffer[(Str.Length/2)-1] ==
| L'\\') {
23464|                 Str.Length-=2;
23465|                 Str.Buffer[(Str.Length/2)]=0;

```

```

23466|         }
23467|         DLOG((TEXT("GVNFVMP='%S'\n"),Buffer));
23468|         goto IsMountMgrName;
23469|     } else {
23470|         DLOG((TEXT("Error %08x getting volume
| for mount point\n"),GetLastError()));
23471|         // fall through
23472|     }
23473| }
23474|
23475| if(_wcsnicmp(Str.Buffer,L"\\device\\",8)==0) {
23476|     UNICODE_STRING Uni={0};
23477|     OBJECT_ATTRIBUTES ObjectAttributes={0};
23478|     HANDLE hVolume;
23479|     IO_STATUS_BLOCK IoStatus={0};
23480|
23481|     RtlInitUnicodeString( &Uni, Str.Buffer);
23482|
23483|     InitializeObjectAttributes (
| &ObjectAttributes,
23484|         &Uni,
23485|
| OBJ_CASE_INSENSITIVE,
23486|         NULL,
23487|         NULL );
23488|
23489|     Err = NtCreateFile( &hVolume,
23490|         FILE_GENERIC_READ,
| // desired access
23491|         &ObjectAttributes,
| // object attributes
23492|         &IoStatus,
23493|         NULL, //
| alloc size
23494|         FILE_ATTRIBUTE_NORMAL,
| // file attributes
23495|         FILE_SHARE_WRITE |
| FILE_SHARE_READ, // share
| access
23496|         FILE_OPEN, //
| create disposition
23497|         | FILE_SYNCHRONOUS_IO_NONALERT, // create
| options
23498|         NULL, // eabuffer
23499|         0 ); // ealength
23500|     if(!Err) {
23501|         Err =
| GetDeviceName(hVolume,Buffer,256);
23502|         NtClose(hVolume);

```

```

23503|
23504|         if(!Err) {
23505|             DLOG((TEXT("Volume supports storage
| get device name\n"))));
23506|             *VolumeType = PSM_IS_SUPPORTED;
23507|             return TRUE;
23508|         } else {
23509|             DLOG((TEXT("Volume does not support
| storage get device name\n"))));
23510|         }
23511|     } else {
23512|         DLOG((TEXT("Error %08x opening
| device\n"),Err));
23513|     }
23514| } else {
23515|     hVolume = CreateFileW(
23516|         VolumeName,
23517|         0,
23518|         FILE_SHARE_READ | FILE_SHARE_WRITE,
23519|         NULL,
23520|         OPEN_EXISTING,
23521|         0,
23522|         NULL );
23523|     if(hVolume!=INVALID_HANDLE_VALUE) {
23524|         Err =
23525|         | GetDeviceName(hVolume,Buffer,256);
23526|         CloseHandle(hVolume);
23527|         // if it supports that ioctl it is a
23528|         | dasd drive
23529|         if(!Err) {
23530|             DLOG((TEXT("Volume supports storage
| get device name\n"))));
23531|             *VolumeType = PSM_IS_SUPPORTED;
23532|             return TRUE;
23533|         } else {
23534|             Err = GetLastError();
23535|             DLOG((TEXT("Error %08x opening
| volume\n"),Err));
23536|             // fall through
23537|         }
23538|     } // of win2k specific stuff
23539|
23540|     DLOG((TEXT("Error %08x determing if volume is local
| hard drive\n"))));
23541|
23542|     *VolumeType = PSM_IS_UNKNOWN;
23543|     // if it failed all of the above.. must not be a

```

```

    | local hard drive
23544|     return FALSE;
23545| }
23546|
23547| STATIC ULONG PSMI_GetSnapShotInfoFromVolume( WCHAR
    | *VolumeName, tSnapShotInfoW *SnapShotInfo )
23548| {
23549|     ULONG Err=ERROR_INVALID_PARAMETER;
23550|     ULONG i;
23551|
23552|     WaitForSingleObject ( SharedMemoryMutex, INFINITE
    | );
23553|
23554|     __try {
23555|
    | for(i=0;i<SharedMemory->NumberOfMappedVolumes;i++) {
23556|         if(
    | (_wcsicmp(SharedMemory->MappedVolumes[i].DriveLetterName
    | ,VolumeName)==0) ||
23557|         | (_wcsicmp(SharedMemory->MappedVolumes[i].VolumeName,Volu
    | meName)==0) ||
23558|         | (_wcsicmp(SharedMemory->MappedVolumes[i].ShareName,Volum
    | eName)==0) ||
23559|         | (_wcsicmp(SharedMemory->MappedVolumes[i].NTDeviceName,Vo
    | lumeName)==0)) {
23560|             if(SnapShotInfo->Size ==
    | sizeof(tSnapShotInfoW)) {
23561|                 pSnapShot Snap =
    | MakePointer(SharedMemory->MappedVolumes[i].SnapShotOffse
    | t);
23562|
23563|         | memmove(&(SnapShotInfo->SnapShot),Snap,sizeof(tSnapShot)
    | );
23564|
23565|         | wcsncpy(SnapShotInfo->Volume
    | ,SharedMemory->MappedVolumes[i].OriginalUserName);
23566|
    | wcsncpy(SnapShotInfo->Company,SharedMemory->MappedVolumes
    | [i].Company);
23567|
    | wcsncpy(SnapShotInfo->Product,SharedMemory->MappedVolumes
    | [i].Product);
23568|         | wcsncpy(SnapShotInfo->Code
    | ,SharedMemory->MappedVolumes[i].Code);
23569|
    | wcsncpy(SnapShotInfo->Version,SharedMemory->MappedVolumes

```

```

    | [i].Version);
23570|         wcsncpy(SnapShotInfo->Key
    | ,SharedMemory->MappedVolumes[i].Key);
23571|         Err = 0;
23572|
23573|         break;
23574|     } else {
23575|         Err = ERROR_INVALID_PARAMETER;
23576|     }
23577| }
23578| }
23579| } __finally {
23580|     ReleaseMutex(SharedMemoryMutex);
23581| }
23582| return Err;
23583| }
23584|
23585|
23586| STATIC ULONG PSMI_OpenManager( HANDLE *hDevice )
23587| {
23588|     TCHAR Name[40]={0};
23589|     ULONG returned=0;
23590|     BOOL B=FALSE;
23591|     ULONG Err=0;
23592|
23593|     | _sprintf(Name,TEXT("\\\\.\\%s_%04x"),TEXT("psman"),
    | PSM_LOW_COMPATIBLE_VERSION );
23594|     *hDevice = CreateFile( Name,
23595|         GENERIC_READ | GENERIC_WRITE,
23596|         FILE_SHARE_READ | FILE_SHARE_WRITE,
23597|         NULL,
23598|         OPEN_EXISTING,
23599|         FILE_FLAG_OVERLAPPED,
23600|         NULL
23601|     );
23602|
23603|     if(*hDevice) {
23604|         Err = 0;
23605|
23606|     } else {
23607|         Err = GetLastError();
23608|     }
23609|
23610|     return Err;
23611| }
23612|
23613|
23614|
23615| STATIC void OpenAPCRoutine(

```

```

23616|  IN pAPCContext ApcContext,
23617|  IN PIO_STATUS_BLOCK IoStatus,
23618|  IN ULONG Reserved
23619|  )
23620| {
23621|  ULONG Err=0;
23622|  WCHAR Str[80];
23623|  pThreadStorage ThreadStorage = GetThreadStorage();
23624|  pSnapShot Snap=NULL;
23625|  tSnapShot SnapTemp;
23626|
23627|  WaitForSingleObject ( OpenCloseMutex, INFINITE );
23628|  __try {
23629|      Err = RtlNtStatusToDosError(IoStatus->Status);
23630|
23631|      DLOG((TEXT("OpenAPCRoutine %08x %08x %08x %08x
| %08x\n"),ApcContext,Err,IoStatus->Status,IoStatus->Infor
| mation,Reserved));
23632|
23633|      if(!Err) {
23634|          // temp so we can alloc the snapshot
23635|          if(!ApcContext->SnapShot) {
23636|              DLOG((TEXT("Snapshot is null, using
| temp\n"))));
23637|              ApcContext->SnapShot = &Snap;
23638|          }
23639|
23640|          /*
23641|          // OTM fossil
23642|          if(ApcContext->OTI->InternalFlags &
| PSM_IFLAG_PERSISTENT) {
23643|              (*ApcContext->SnapShot) = &SnapTemp;
23644|          } else {
23645|              (*ApcContext->SnapShot) =
| AllocSnapShot();
23646|          }
23647|          */
23648|          (*ApcContext->SnapShot) = &SnapTemp;
23649|
23650|          if((*ApcContext->SnapShot)) {
23651|              (*ApcContext->SnapShot)->SnapShotTime =
| ApcContext->OTO->SnapShotTime;
23652|
| (*ApcContext->SnapShot)->KernelSnapShotPointer =
| ApcContext->OTO->KernelSnapShotPointer;
23653|              (*ApcContext->SnapShot)->OwningThread =
| ThreadStorage;
23654|              (*ApcContext->SnapShot)->Instance =
| ApcContext->OTO->Instance;
23655|

```

```

23656|         if(ApcContext->OTI->InternalFlags &
| PSM_IFLAG_PERSISTENT) {
23657|         | (*ApcContext->SnapShot)->SnapShotType =
| PSM_SS_TYPE_PERSISTENT;
23658|             ThreadStorage->Persistent = TRUE;
23659|         } else {
23660|         | (*ApcContext->SnapShot)->SnapShotType =
| PSM_SS_TYPE_TEMPORARY;
23661|             ThreadStorage->Persistent = FALSE;
23662|         }
23663|
23664|         DLOG((TEXT("sst=%l64x, kssp=%08x,
| ts=%08x,
| i=%d\n"),ApcContext->OTO->SnapShotTime,ApcContext->OTO->
| KernelSnapShotPointer,ThreadStorage,ApcContext->OTO->Ins
| tance));
23665|
23666|         Err =
| AddDrivesToSystemPersistent(ApcContext,&ApcContext->OTO-
| >SnapShotTime);
23667|         (*ApcContext->SnapShot) =
| ApcContext->OTO->KernelSnapShotPointer;
23668|         ApcContext->SnapShot = NULL;
23669|
| UpdateClusterRegistries(ApcContext->OTO->KernelSnapShotP
| ointer);
23670|         /*
23671|         if(!ThreadStorage->Persistent) {
23672|         Err =
| AddDrivesToSystem(ApcContext);
23673|         if(!Err) {
23674|             ThreadStorage->NumOpens++;
23675|
| ThreadStorage->SnapShotsOffset[ThreadStorage->NumSnapSho
| ts++] = MakeOffset((*ApcContext->SnapShot));
23676|
23677|             WaitForSingleObject (
| SharedMemoryMutex, INFINITE );
23678|             __try {
23679|                 DLOG(("Adding Snapshot %08x
| %d (%08x) to
| list\n",(*ApcContext->SnapShot),(*ApcContext->SnapShot)-
| >Instance,(*ApcContext->SnapShot)->KernelSnapShotPointer
| ));
23680|
| SharedMemory->SnapShotsOffset[SharedMemory->NumberOfSnap
| Shots++] = MakeOffset((*ApcContext->SnapShot));
23681|             } __finally {

```

```

23682|
| ReleaseMutex(SharedMemoryMutex);
23683|     }
23684|     if((* (DWORD*)ApcContext->Out)
| == 0x5a4b3c2d) {
23685|         // skip over \DosDevices\
| we added
23686|
| CharToOemW(&ApcContext->OTO->CacheFileName[12],(CHAR*)Ap
| cContext->Out->CacheFileName);
23687|     } else {
23688|
| wcsncpy((WCHAR*)ApcContext->Out->CacheFileName,&ApcContex
| t->OTO->CacheFileName[12]);
23689|     }
23690|
23691|     }
23692|
23693|         // some error occured having our
| volumes mapped in
23694|         // (maybe out of drive letters ;)
23695|         if(Err) {
23696|             HANDLE hEvent = CreateEvent(
| NULL, FALSE, FALSE, NULL );
23697|             // unable to add volumes to
| system, error out.
23698|             DLOG(("Error %08x during drive
| adding, calling close\n",Err));
23699|
| PSMI_Close_Internal(ThreadStorage->PSManHandle,hEvent,Ap
| cContext->OTO->KernelSnapShotPointer);
23700|             CloseHandle(hEvent);
23701|             IoStatus->Status = Err;
23702|
| FreeSnapShot((*ApcContext->SnapShot));
23703|         }
23704|
23705|     } else { // if !persistent
23706|         Err =
| AddDrivesToSystemPersistent(ApcContext,&ApcContext->OTO-
| >SnapShotTime);
23707|         (*ApcContext->SnapShot) =
| ApcContext->OTO->KernelSnapShotPointer;
23708|         ApcContext->SnapShot = NULL;
23709|
| UpdateClusterRegistries(ApcContext->OTO->KernelSnapShotP
| ointer);
23710|     }
23711|     */
23712| } else {

```



```

23713|         HANDLE hEvent = CreateEvent( NULL,
| FALSE, FALSE, NULL );
23714|         // out of memory?
23715|         // unable to add volumes to system,
| error out.
23716|         DLOG(("Error %08x during memory alloc
| for snapshot\n",GetLastError()));
23717|         | PSMI_Close_Internal(ThreadStorage->PSManHandle,hEvent,Ap
| cContext->OTO->KernelSnapShotPointer);
23718|         CloseHandle(hEvent);
23719|         Err = IoStatus->Status =
| ERROR_OUTOFMEMORY;
23720|     }
23721| } else {
23722|     // delete cache file if ANY error occurred.
| This is to keep zero length files
23723|     // from hanging around... one day we may
| want to use another function than
23724|     // GetTempFile so we do not make these temp
| files.
23725|     // only delete cache file if it was a temp
| and not permanent
23726|     DLOG(("Error %08x during open\n",Err));
23727|     if(ThreadStorage->UsedTempFile) {
23728|         DLOG((TEXT("Deleting cache file due to
| error %08x\n"),IoStatus->Status));
23729|         | DeleteFileW(ApcContext->In.CacheFileName);
23730|     }
23731| }
23732|
23733|     swprintf(Str,L"%08x",Err);
23734|     // execute postopen file
23735|
| ExecuteFile(PostOpenFile,Str,ChildUser,ChildPassword);
23736|
23737|     // set last error now that we are done with all
| routines that will access it.
23738|     SetLastError(Err);
23739|
23740|     if(ApcContext->Overlapped) {
23741|         ApcContext->Overlapped->Internal =
| IoStatus->Status;
23742|         ApcContext->Overlapped->InternalHigh = Err;
23743|
| if(ApcContext->Overlapped->hEvent!=INVALID_HANDLE_VALUE)
| {
23744|         | SetEvent(ApcContext->Overlapped->hEvent);

```

```

23745|     }
23746| }
23747|
23748|     if(ApcContext->OverlappedRoutine) {
23749|         __try {
23750|
23751|             | (ApcContext->OverlappedRoutine)(Err,IoStatus->Informatio
23752|             | n,ApcContext->Overlapped);
23753|         }
23754|         | __except(ExceptionFilter(GetExceptionInformation())) {
23755|             DLOG((TEXT("Exception %08x in apc
23756|             | call\n"),GetExceptionCode()));
23757|         }
23758|     }
23759|
23760|     LocalFree(ApcContext->VolumeMapFlags);
23761|     LocalFree(ApcContext->OTI);
23762|     LocalFree(ApcContext->OTO);
23763|     LocalFree(ApcContext);
23764| } __finally {
23765|     ReleaseMutex(OpenCloseMutex);
23766| }
23767|
23768| return;
23769| }
23770|
23771| STATIC LONG SetFlushRoutineForDriver( HANDLE hDevice )
23772| {
23773|     ULONG B=FALSE;
23774|     tSetFlushRoutine Flush;
23775|     LONG Err;
23776|     OVERLAPPED o;
23777|     ULONG returned;
23778|
23779|     Flush.ZwFlushBuffersFile =
23780|         | (void*)GetProcAddress(GetModuleHandle("ntdll.dll"),"ZwFI
23781|         | ushBuffersFile");
23782|     //Flush.ZwFlushBuffersFile =
23783|         | (void*)GetProcAddress(GetModuleHandle("ntdll.dll"),"_imp
23784|         | __ZwFlushBuffersFile");
23785|
23786|     if(Flush.ZwFlushBuffersFile) {
23787|         memset(&o,0,sizeof(o));
23788|         o.hEvent = CreateEvent( NULL, FALSE, FALSE,
23789|         | NULL );
23790|
23791|         B = DeviceIoControl( hDevice,

```

```

23786|     IOCTL_SET_FLUSH_ROUTINE,
23787|     &Flush,
23788|     sizeof(Flush),
23789|     NULL,
23790|     0,
23791|     &returned,
23792|     &o);
23793|
23794|     if(B) {
23795|         Err = 0;
23796|     } else {
23797|         Err = GetLastError();
23798|         DLOG((TEXT("Error %08x Calling set flush
| routine\n"),Err));
23799|     }
23800|     CloseHandle(o.hEvent);
23801| } else {
23802|     Err = GetLastError();
23803|     DLOG((TEXT("Error %08x getting entry point for
| flush\n"),Err));
23804| }
23805| return Err;
23806| }
23807|
23808| STATIC ULONG PSMI_CommonOpen(
23809|     HANDLE hDevice,
23810|     IN pOpenTransactionIn3W In,
23811|     IN  ULONG NumVolumes,
23812|     IN  PVOID InVolumeMap,
23813|     IN  ULONG *VolumeMapFlags,
23814|     IN  ULONG OutVolumeMapByteSize,
23815|     INOUT PVOID OutVolumeMap,
23816|     OUT pOpenTransactionOutW Out,
23817|     INOUT LPOVERLAPPED Overlapped,
23818|     IN LPOVERLAPPED_COMPLETION_ROUTINE
| CompletionRoutine,
23819|     IN ULONG InternalFlags,
23820|     OUT pSnapShot *SnapShot
23821| )
23822| {
23823|     ULONG Err=0;
23824|     pThreadStorage ThreadStorage = GetThreadStorage();
23825|     tAPCContext *APCContext;
23826|     WCHAR Str[80];
23827|     HKEY Key;
23828|     ULONG DataSize;
23829|     TCHAR RegStr[255];
23830|     ULONG InternalBufferChars = 0;
23831|
23832|     if((!In) || (!Out) || (!InVolumeMap) ||

```

```

    | (!VolumeMapFlags) || (!OutVolumeMap) ||
    | (OutVolumeMapByteSize<8) || ((signed)NumVolumes<1)) {
23833|         return ERROR_INVALID_PARAMETER;
23834|     }
23835|
23836|     // make sure they are passing in something we know
    | about
23837|     if(
23838|         (In->Size != sizeof(tOpenTransactionIn3W)) &&
23839|         (In->Size != sizeof(tOpenTransactionIn1W)) &&
23840|         (In->Size != sizeof(tOpenTransactionIn2W))
23841|     ) {
23842|         return ERROR_INVALID_PARAMETER;
23843|     }
23844|
23845|     // check the length so we dont copy over the stack
    | and destory
23846|     // everything we are lookin at if the things
    | contain long names
23847|     if(
    | !CheckForZeroTerminatorW(In->CacheFileName,256)) {
23848|         return ERROR_INVALID_PARAMETER;
23849|     }
23850|     if (!CheckForZeroTerminatorW(In->PreOpenFile,256))
    | ||
23851|         (!CheckForZeroTerminatorW(In->PostOpenFile,256))
    | ||
23852|
    | (!CheckForZeroTerminatorW(In->PostCloseFile,256)) ||
23853|         (!CheckForZeroTerminatorW(In->UserName,40)) ||
23854|         (!CheckForZeroTerminatorW(In->Password,40)) ) {
23855|         return ERROR_INVALID_PARAMETER;
23856|     }
23857|
23858| /* //OTM fossil
23859|     if(!(In->Flags & PSM_FLAG_PERSISTENT_SNAPSHOT)) {
23860|         // not used in persistent snapshots
23861|
23862|         // 1 terabyte
23863|         if(In->SizeOfCacheFileMB > 1 * 1024 * 1024) {
23864|             return ERROR_INVALID_PARAMETER;
23865|         }
23866|         if(In->MaxSizeOfCacheFileMB > 1 * 1024 * 1024)
    | {
23867|             return ERROR_INVALID_PARAMETER;
23868|         }
23869|         if(In->SizeOfCacheFileMB >
    | In->MaxSizeOfCacheFileMB) {
23870|             return ERROR_INVALID_PARAMETER;
23871|         }

```

```

23872|    // 1 hour
23873|    if(In->QuiescentWait > 1 * 60 * 60) {
23874|        return ERROR_INVALID_PARAMETER;
23875|    }
23876|    // 1 week
23877|    if(In->QuiescentTimeout > 1 * 7 * 24 * 60 * 60)
    | {
23878|        return ERROR_INVALID_PARAMETER;
23879|    }
23880| }
23881| */
23882|
23883| if((Overlapped) &&
    | (Overlapped->hEvent!=INVALID_HANDLE_VALUE))
23884|     ResetEvent(Overlapped->hEvent);
23885|
23886| APCContext = LocalAlloc(LPTR,sizeof(tAPCContext));
23887| if(APCContext) {
23888|
23889|     // keep this read only...
23890|     memmove(&APCContext->In,In,sizeof(*In));
23891|
23892|     // 4 bytes per pointer, 100 bytes (by default)
    | of "string space"
23893|     InternalBufferChars = 100 * NumVolumes;
23894|     APCContext->SizeofOTI =
    | sizeof(tOpenTransactionInInternal)+(NumVolumes*4)+(Inter
    | nalBufferChars*sizeof(WCHAR));
23895|     APCContext->SizeofOTO =
    | sizeof(tOpenTransactionOutInternal)+(NumVolumes*4)+(Inte
    | rnalBufferChars*sizeof(WCHAR));
23896|     APCContext->OTI =
    | LocalAlloc(LPTR,APCContext->SizeofOTI);
23897|     APCContext->OTO =
    | LocalAlloc(LPTR,APCContext->SizeofOTO);
23898|     APCContext->VolumeMapFlags =
    | LocalAlloc(LPTR,NumVolumes*sizeof(ULONG));
23899|
23900|     if((!APCContext->OTI) || (!APCContext->OTO) ||
    | (!APCContext->VolumeMapFlags)) {
23901|         Err = ERROR_OUTOFMEMORY;
23902|         goto ErrorExit;
23903|     }
23904|
    | memset(APCContext->OTI,0,APCContext->SizeofOTI);
23905|
    | memset(APCContext->OTO,0,APCContext->SizeofOTO);
23906|
    | memcpy(APCContext->VolumeMapFlags,VolumeMapFlags,NumVolu
    | mes*sizeof(ULONG));

```

```

23907|
23908|     if((In->Size>=sizeof(tOpenTransactionIn2W)) &&
| (In->SecurityInfo)) {
23909|         | if(In->SecurityInfo->Size==sizeof(tPSM_SecurityInfo)) {
23910|             | memcpy(&APCContext->SecurityInfo,In->SecurityInfo,sizeof
| (tPSM_SecurityInfo));
23911|             } else {
23912|                 Err = ERROR_INVALID_PARAMETER;
23913|                 goto ErrorExit;
23914|             }
23915|         } else {
23916|             APCContext->SecurityInfo.Size =
| sizeof(tPSM_SecurityInfo);
23917|             | wcsncpy(APCContext->SecurityInfo.Remark,L"PSM Shared
| Drive");
23918|             APCContext->SecurityInfo.MaxUses =
| SHI_USES_UNLIMITED;
23919|             APCContext->SecurityInfo.Permissions =
| ACCESS_READ | ACCESS_EXEC;
23920|             APCContext->SecurityInfo.SecurityDescriptor
| = NULL;
23921|         }
23922|     }
23923|     {
23924|         ULONG i;
23925|         for(i=0;i<NumVolumes;i++) {
23926|             DLOG((TEXT("Flags %d (%08x) =
| %08x:%08x\n"),i,APCContext->VolumeMapFlags,APCContext->V
| olumeMapFlags[i],VolumeMapFlags[i]));
23927|         }
23928|     }
23929|
23930|     APCContext->SizeOfVolumeMap =
| OutVolumeMapByteSize;
23931|     APCContext->VolumeMap = OutVolumeMap;
23932|     APCContext->Out = Out;
23933|     APCContext->Overlapped = Overlapped;
23934|     APCContext->OverlappedRoutine =
| CompletionRoutine;
23935|     APCContext->SnapShot = SnapShot;
23936|
23937|     APCContext->OTI->Size =
| sizeof(tOpenTransactionInInternal);
23938|     APCContext->OTI->SizeOfCacheFileMB =
| In->SizeOfCacheFileMB;
23939|     APCContext->OTI->MaxSizeOfCacheFileMB =
| In->MaxSizeOfCacheFileMB;

```

```

23940|     APCContext->OTI->Flags          =
| In->Flags;
23941|     APCContext->OTI->QuiescentWait    =
| In->QuiescentWait;
23942|     APCContext->OTI->QuiescentTimeout =
| In->QuiescentTimeout;
23943|     APCContext->OTI->AbortEvent       =
| In->AbortEvent;
23944|     APCContext->OTI->InternalFlags    =
| InternalFlags;
23945|     APCContext->OTI->CallerPrivateUse =
| In->CallerPrivateUse;
23946|     APCContext->OTI->NumToKeep        =
| In->NumToKeep;
23947|     APCContext->OTI->Priority =
| ((tOpenTransactionInPersistentW*)In)->Priority;
23948|     APCContext->OTI->SnapShotFlags =
| ((tOpenTransactionInPersistentW*)In)->SnapShotFlags;
23949|
23950|     // we dont have a need for this yet..
23951|     APCContext->OTI->DllPrivateUse    = NULL;
23952|
23953|     if(In->Flags & PSM_FLAG_PERSISTENT_SNAPSHOT) {
23954|         APCContext->OTI->InternalFlags |=
| PSM_IFLAG_PERSISTENT;
23955|     }
23956|
23957|     if(In->Flags & PSM_FLAG_SAVE_TEMP_ON_EXIT) {
23958|         APCContext->OTI->InternalFlags |=
| PSM_IFLAG_SAVE_TEMP_ON_EXIT;
23959|     }
23960|
23961|     // exchange the two if backwards
23962|
| if(APCContext->OTI->QuiescentWait>APCContext->OTI->Quies
| centTimeout) {
23963|         ULONG Save =
| APCContext->OTI->QuiescentTimeout;
23964|         APCContext->OTI->QuiescentTimeout =
| APCContext->OTI->QuiescentWait;
23965|         APCContext->OTI->QuiescentWait = Save;
23966|     }
23967|
23968|     ThreadStorage->AbortEvent = In->AbortEvent;
23969|
23970|     // do volumemap conversion here.
23971|
23972|     Err = MakeVolumeList (
23973|         NumVolumes,
23974|         InVolumeMap,

```

```

23975|         OutVolumeMapByteSize,
23976|         APCCContext->OTI,
23977|         InternalBufferChars );
23978|
23979|     if ( Err != 0 ) {
23980|         // specified nothing to do.
23981|         goto ErrorExit;
23982|     }
23983|
23984|     APCCContext->OTI->NumberOfDevices = NumVolumes;
23985|
23986|     if(APCCContext->OTI->NumberOfDevices>MAX_VOLUMES_SUPPORTE
23987|     | D) {
23988|         Err = ERROR_OUT_OF_STRUCTURES;
23989|         goto ErrorExit;
23990|     }
23991|
23992|     if(wcsncmp(In->CacheFileName,L"")==0) {
23993|         WCHAR TempPath[255];
23994|
23995|         if(CacheFileLocationKeyExists) {
23996|             if(wcsncmp(CacheFileLocation,L"")==0) {
23997|                 // Empty registry key, pick file in
23998|                 | temp directory
23999|                 GetTempPathW(255,TempPath);
24000|             } else {
24001|                 // okay use path specified
24002|                 wcsncpy(TempPath,CacheFileLocation);
24003|             }
24004|         } else {
24005|             // key doesnt exist so lets find a
24006|             | suitable volume
24007|             // for our cache file, preferably one
24008|             | not being
24009|             // psmed
24010|             | FindBestVolumeForCache(In,TempPath,sizeof(TempPath)/size
24011|             | of(TempPath[0]));
24012|         }
24013|
24014|         GetTempFileNameW(
24015|         TempPath, // pointer to directory name
24016|         | for temporary file
24017|         L"PSM", // pointer to filename
24018|         | prefix
24019|         ThreadStorage->NumOpens, //
24020|         | number used to create temporary filename
24021|         ThreadStorage->TempFileName);
24022|         ThreadStorage->UsedTempFile = TRUE;

```



```

24015|
    | swprintf(APCContext->OTI->CacheFileName,L"\\DosDevices\\
    | %s",ThreadStorage->TempFileName);
24016|     } else {
24017|         // must be in unicode...
24018|
    | swprintf(APCContext->OTI->CacheFileName,L"\\DosDevices\\
    | %s",In->CacheFileName);
24019|     ThreadStorage->UsedTempFile = FALSE;
24020|     }
24021|
24022|     #if 0
24023|         if(DebugMode)
24024|
    | wprintf(L""%s\\n",APCContext->OTI->CacheFileName);
24025|     #endif
24026|
24027|     // if a temp file we created, always delete it
24028|     if(ThreadStorage->UsedTempFile) {
24029|         APCContext->OTI->Flags &=
    | ~PSM_FLAG_DO_NOT_DELETE_CACHE;
24030|     }
24031|     // map errorcallback here
24032|     APCContext->OTI->ErrorEvent      =
    | In->ErrorEvent;
24033|
24034|     // they specified batch files
24035|     wcscpy(PreOpenFile,In->PreOpenFile);
24036|     if(wcscmp(PreOpenFile,L"")==0)
24037|         wcscpy(PreOpenFile,L"preopen");
24038|
24039|     wcscpy(PostOpenFile,In->PostOpenFile);
24040|     if(wcscmp(PostOpenFile,L"")==0)
24041|         wcscpy(PostOpenFile,L"postopen");
24042|
24043|     wcscpy(PostCloseFile,In->PostCloseFile);
24044|     if(wcscmp(PostCloseFile,L"")==0)
24045|         wcscpy(PostCloseFile,L"postclse");
24046|
24047|     wcscpy(ChildUser,In->UserName);
24048|     wcscpy(ChildPassword,In->Password);
24049|     if(wcscmp(ChildUser,L"")==0) {
24050|         wcscpy(ChildUser,L"");
24051|         wcscpy(ChildPassword,L"");
24052|
24053|
    | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
    | es\\PSMan%d"), NtBuildNumber > 1381 ? 5 : 4);
24054|     // open the registry
24055|     Err = RegOpenKeyEx(

```

```

24056|         HKEY_LOCAL_MACHINE, // handle of open
      | key
24057|         RegStr, // address of name of subkey to
      | open
24058|         0, // reserved
24059|         KEY_READ, // security access mask
24060|         &Key// address of handle of open key
24061|     );
24062|     if(Err==0) {
24063|
24064|         DataSize = 256;
24065|         Err = RegQueryValueExW(
24066|             Key, // handle of key to query
24067|             L"ChildUser", // address of name
      | of value to query
24068|             NULL, // reserved
24069|             NULL, // address of buffer for
      | value type
24070|             (char*)ChildUser, // address of
      | data buffer
24071|             &DataSize // address of data
      | buffer size
24072|         );
24073|
24074|         DataSize = 256;
24075|         RegQueryValueExW(
24076|             Key, // handle of key to query
24077|             L"ChildPassword", // address of
      | name of value to query
24078|             NULL, // reserved
24079|             NULL, // address of buffer for
      | value type
24080|             (char*)ChildPassword, // address
      | of data buffer
24081|             &DataSize // address of data
      | buffer size
24082|         );
24083|
24084|         // close the registry
24085|         RegCloseKey(Key);
24086|     }
24087| }
24088|
24089| // execute preopen file
24090|
      | ExecuteFile(PreOpenFile,L"",ChildUser,ChildPassword);
24091|
24092|     APCContext->IoStatus.Status=0;
24093|     APCContext->IoStatus.Information=0;
24094|

```

```

24095|         if(hDevice) {
24096|
24097|             // make sure it has access to flush routine
24098|             // we have to do this as ntoskrnl doesnt
        | export Nt/ZwFlushBuffersFile
24099|             // or any routines to dynamically import
        | the routine
24100|             Err = SetFlushRoutineForDriver(hDevice);
24101|             if(Err) {
24102|                 goto BadBadExit;
24103|             }
24104|
24105|             Err = NtDeviceIoControlFile(
24106|                 hDevice,
24107|                 NULL,
24108|                 OpenAPCRoutine,
24109|                 APCCContext,
24110|                 &APCCContext->IoStatus,
24111|                 IOCTL_TURNON_PSM,
24112|                 APCCContext->OTI,
24113|                 APCCContext->SizeofOTI,
24114|                 APCCContext->OTO,
24115|                 APCCContext->SizeofOTO
24116|             );
24117|             if(Err==0x103) {
24118|                 Overlapped->Internal = 0x103; //
        | status_pending
24119|                 Err = ERROR_IO_PENDING;
24120|             } else {
24121|                 DLOG((TEXT("Error %08x (%08x) during
        | open\n"),Err,GetLastError()));
24122|                 Err = RtlNtStatusToDosError(Err);
24123|                 // callback not called if error
        | occurred.
24124|                 if(Err) {
24125| BadBadExit:
24126|                     // delete cache file if created
24127|                     if(ThreadStorage->UsedTempFile) {
24128|                         DeleteFileW(ThreadStorage->TempFileName);
24129|                     }
24130|                     swprintf(Str,L"%08x",Err);
24131|                     // execute postopen file
24132|
        | ExecuteFile(PostOpenFile,Str,ChildUser,ChildPassword);
24133| ErrorExit:
24134|                     if(APCCContext->VolumeMapFlags)
        | LocalFree(APCCContext->VolumeMapFlags);
24135|                     if(APCCContext->OTI)
        | LocalFree(APCCContext->OTI);

```

```

24136|         if(APCContext->OTO)
| LocalFree(APCContext->OTO);
24137|         LocalFree(APCContext);
24138|     }
24139| }
24140| } else {
24141|     Err = ERROR_INVALID_HANDLE;
24142|     goto BadBadExit;
24143| }
24144| } else {
24145|     Err = ERROR_OUTOFMEMORY;
24146| }
24147|
24148| if(Err) {
24149|     SetLastError(Err);
24150| }
24151|
24152| return Err;
24153| }
24154|
24155|
24156| STATIC ULONG PSMI_OpenEx(
24157|     HANDLE hDevice,
24158|     IN pOpenTransactionIn3W In,
24159|     IN  ULONG NumVolumes,
24160|     IN  PVOID InVolumeMap,
24161|     IN  ULONG *VolumeMapFlags,
24162|     IN  ULONG OutVolumeMapByteSize,
24163|     INOUT PVOID OutVolumeMap,
24164|     OUT pOpenTransactionOutW Out,
24165|     INOUT LPOVERLAPPED Overlapped,
24166|     IN LPOVERLAPPED_COMPLETION_ROUTINE
| CompletionRoutine,
24167|     IN ULONG InternalFlags,
24168|     OUT pSnapShot *SnapShot
24169| )
24170| {
24171|     return
| PSMI_CommonOpen(hDevice,In,NumVolumes,InVolumeMap,Volume
| MapFlags,OutVolumeMapByteSize,OutVolumeMap,Out,Overlapped,
| CompletionRoutine,InternalFlags,SnapShot);
24172| }
24173|
24174| STATIC ULONG PSMI_GetVersion ( HANDLE hDevice, HANDLE
| hEvent, ULONG VerSize, pPSM_VersionInfo Version )
24175| {
24176|     BOOL B=FALSE;
24177|     ULONG Err=0;
24178|     ULONG returned=0;
24179|     OVERLAPPED o;

```

```

24180|
24181|     if(hDevice) {
24182|         memset(&o,0,sizeof(o));
24183|         ResetEvent(hEvent);
24184|         o.hEvent = hEvent;
24185|
24186|         B = DeviceIoControl( hDevice,
24187|             IOCTL_GET_VERSION,
24188|             &VerSize,
24189|             sizeof(ULONG),
24190|             Version,
24191|             sizeof(tPSM_VersionInfo),
24192|             &returned,
24193|             &o);
24194|
24195|         if (B) {
24196|             Err = 0;
24197|         } else {
24198|             Err = GetLastError();
24199|             if (Err == ERROR_IO_PENDING) {
24200|                 WaitForSingleObject ( hEvent, INFINITE
24201|                     | );
24202|                 Err = GetLastError();
24203|             }
24204|         } else {
24205|             Err = ERROR_INVALID_HANDLE;
24206|         }
24207|
24208|     return Err;
24209|
24210| }
24211|
24212| ULONG PSMI_GetVolumeStats( HANDLE hDevice, HANDLE
24213|     | hEvent, WCHAR *VolumeName, tPSM_GetStatsRecord *Stats )
24214| {
24215|     BOOL B=FALSE;
24216|     ULONG Err=0;
24217|     ULONG returned=0;
24218|     OVERLAPPED o;
24219|     WCHAR NewName[256];
24220|
24221|     | if((Err=GetNTDeviceName(VolumeName,NewName,256,FALSE))==
24222|     | 0) {
24223|         if(hDevice) {
24224|             memset(&o,0,sizeof(o));
24225|             ResetEvent(hEvent);
24226|             o.hEvent = hEvent;

```

```

24226|         B = DeviceIoControl( hDevice,
24227|             IOCTL_GET_VOLUME_STATS,
24228|             NewName,
24229|             | wcslen(NewName)*sizeof(WCHAR)+sizeof(WCHAR),
24230|             Stats,
24231|             sizeof(tPSM_GetStatsRecord),
24232|             &returned,
24233|             &o);
24234|
24235|         if (B) {
24236|             Err = 0;
24237|         } else {
24238|             Err = GetLastError();
24239|             if (Err == ERROR_IO_PENDING) {
24240|                 WaitForSingleObject ( hEvent,
24241|                     | INFINITE );
24242|                 Err = GetLastError();
24243|             }
24244|         } else {
24245|             Err = ERROR_INVALID_HANDLE;
24246|         }
24247|     }
24248|
24249|     return Err;
24250|
24251| }
24252|
24253| ULONG PSMI_GetVolumeInfo( HANDLE hDevice, HANDLE
24254|     | hEvent, tPSMVolumeInfoIn *In, tPSMVolumeInfoOut *Out )
24255| {
24256|     BOOL B=FALSE;
24257|     ULONG Err=0;
24258|     ULONG returned=0;
24259|     OVERLAPPED o;
24260|     if(hDevice) {
24261|         memset(&o,0,sizeof(o));
24262|         ResetEvent(hEvent);
24263|         o.hEvent = hEvent;
24264|
24265|         B = DeviceIoControl( hDevice,
24266|             IOCTL_GET_VOLUME_INFO,
24267|             In,
24268|             sizeof(tPSMVolumeInfoIn),
24269|             Out,
24270|             sizeof(tPSMVolumeInfoOut),
24271|             &returned,
24272|             &o);

```

```

24273|
24274|     if (B) {
24275|         Err = 0;
24276|     } else {
24277|         Err = GetLastError();
24278|         if (Err == ERROR_IO_PENDING) {
24279|             WaitForSingleObject ( hEvent, INFINITE
24280| | );
24281|             Err = GetLastError();
24282|         }
24283|     } else {
24284|         Err = ERROR_INVALID_HANDLE;
24285|     }
24286|
24287|     return Err;
24288| }
24289|
24290| STATIC ULONG GenHash( WCHAR *String )
24291| {
24292|     ULONG HashVal = 0;
24293|
24294|     while ( *String!= L'\0' ) {
24295|         HashVal = ( HashVal << 5 ) + *String++;
24296|     }
24297|     return HashVal;
24298| }
24299|
24300| STATIC ULONG LastVolClusterSize=0;
24301| STATIC ULONG LastVolClusterOffset=0;
24302| STATIC ULONG LastVolHash=0;
24303|
24304| ULONG GetVolumeClusterSize( HANDLE hDevice, HANDLE
24305| | hEvent, WCHAR *VolumeName, ULONG *SPC, ULONG *C0Offset
24306| | )
24307| {
24308|     ULONG BPS=512;
24309|     WCHAR New[256];
24310|     HANDLE FileHandle;
24311|     UNICODE_STRING Uni;
24312|     FILE_FS_SIZE_INFORMATION Fs;
24313|     ULONG Err;
24314|     OBJECT_ATTRIBUTES ObjectAttributes;
24315|     IO_STATUS_BLOCK Io;
24316|     tPSMVolumeInfoIn In;
24317|     tPSMVolumeInfoOut Out;
24318|
24319|     wcscpy(New,VolumeName);
24320|     wscat(New,L"\\");

```

```

24320|   if(GenHash(New)==LastVolHash) {
24321|       Err=0;
24322|       goto Match;
24323|   }
24324|
24325|   LastVolHash = GenHash(New);
24326|
24327|   RtlInitUnicodeString( &Uni, New);
24328|
24329|   InitializeObjectAttributes ( &ObjectAttributes,
24330|                               &Uni,
24331|                               OBJ_CASE_INSENSITIVE,
24332|                               NULL,
24333|                               NULL );
24334|
24335|   Err = NtCreateFile( &FileHandle,
24336|                      FILE_GENERIC_READ,
24337|                      | // desired access
24338|                      &ObjectAttributes,
24339|                      | // object attributes
24340|                      &Io,
24341|                      NULL,          //
24342|                      | alloc size
24343|                      FILE_ATTRIBUTE_NORMAL,
24344|                      | // file attributes
24345|                      FILE_SHARE_WRITE |
24346|                      | FILE_SHARE_READ,          // share
24347|                      | access
24348|                      FILE_OPEN,          //
24349|                      | create disposition
24350|                      | FILE_NO_INTERMEDIATE_BUFFERING |
24351|                      | FILE_SYNCHRONOUS_IO_NONALERT |
24352|                      | FILE_OPEN_FOR_BACKUP_INTENT,          // create
24353|                      | options
24354|                      NULL, // eabuffer
24355|                      0 ); // ealength
24356|
24357|
24358|
24359|
24360|   if(!Err) {
24361|       if(pNtQueryVolumeInformation) {
24362|           Err =
24363|               | pNtQueryVolumeInformation(FileHandle,&Io,&Fs,sizeof(Fs),
24364|               | FileFsSizeInformation);
24365|
24366|           if(!Err) {
24367|               BPS = Fs.BytesPerSector;
24368|               *SPC = Fs.SectorsPerAllocationUnit;
24369|           } else {

```



```

24357|         DLOG((TEXT("Error %08x querying vol
| info\n"),Err));
24358|         DebugBreak();
24359|     }
24360| } else {
24361|     DLOG((TEXT("NtQuery doesnt exist\n"),Err));
24362|     DebugBreak();
24363| }
24364|     NtClose(FileHandle);
24365| } else {
24366|     DLOG((TEXT("Error %08x opening directory
| '%S'\n"),Err,New));
24367|     DebugBreak();
24368| }
24369| LastVolClusterSize = *SPC;
24370|
24371| if(!Err) {
24372|     GetNTDeviceName(VolumeName,New,256,FALSE);
24373|
24374|     In.Size = sizeof(tPSMVolumeInfoIn);
24375|     wcscpy(In.VolumeName,New);
24376|
24377|     Err = PSMI_GetVolumeInfo( hDevice,hEvent, &In,
| &Out );
24378|
24379|     if(!Err) {
24380|         LastVolCluster0Offset = Out.Cluster0Offset;
24381|     } else {
24382|         LastVolCluster0Offset = 0;
24383|     }
24384| }
24385| Match:
24386| *C0Offset = LastVolCluster0Offset;
24387| *SPC = LastVolClusterSize;
24388| return Err;
24389| }
24390|
24391| STATIC ULONG PSMI_FreeRanges ( HANDLE hDevice, HANDLE
| hEvent, tPSM_FreeRanges *Ranges )
24392| {
24393|     BOOL B=FALSE;
24394|     ULONG Err=0;
24395|     ULONG returned=0;
24396|     OVERLAPPED o;
24397|
24398|     if(hDevice) {
24399|         memset(&o,0,sizeof(o));
24400|         ResetEvent(hEvent);
24401|         o.hEvent = hEvent;
24402|

```

```

24403|     B = DeviceIoControl( hDevice,
24404|         IOCTL_FREE_RANGES,
24405|         Ranges,
24406|         | sizeof(tPSM_FreeRanges)+sizeof(tPSM_RangeList)*Ranges->N
          | umberOfRanges,
24407|         NULL,
24408|         0,
24409|         &returned,
24410|         &o);
24411|
24412|     if (B) {
24413|         Err = 0;
24414|     } else {
24415|         Err = GetLastError();
24416|         if (Err == ERROR_IO_PENDING) {
24417|             WaitForSingleObject ( hEvent, INFINITE
          | );
24418|             Err = GetLastError();
24419|         }
24420|     }
24421| } else {
24422|     Err = ERROR_INVALID_HANDLE;
24423| }
24424|
24425| return Err;
24426| }
24427|
24428| STATIC ULONG PSMI_FreeFiles( HANDLE hDevice, HANDLE
          | hEvent, pSnapShot SnapShot, ULONG NumberOfFiles, WCHAR
          | *Files[] )
24429| {
24430|     ULONG Err=0;
24431|     ULONG SaveErr=0;
24432|     ULONG i=0, j=0;
24433|     tPSM_FreeRanges *Ranges=0;
24434|     RETRIEVAL_POINTERS_BUFFER *RP=NULL;
24435|     STARTING_VCN_INPUT_BUFFER SVIB={0};
24436|     ULONG RPSize=0;
24437|     HANDLE FileHandle=INVALID_HANDLE_VALUE;
24438|     ULONG Count=0;
24439|     LARGE_INTEGER StartVcn={0};
24440|     ULONG VolumeClusterSize;
24441|     WCHAR VirtualDrive[256];
24442|     WCHAR OriginalDrive[256];
24443|     WCHAR Temp[256];
24444|     WCHAR *FileNameOnly=0;
24445|     LARGE_INTEGER LI={0};
24446|     ULONG FileLen=0;
24447|     FILE_STANDARD_INFORMATION FSInfo;

```

```

24448| FILE_BASIC_INFORMATION FBInfo;
24449| UNICODE_STRING Uni;
24450| OBJECT_ATTRIBUTES ObjectAttributes;
24451| IO_STATUS_BLOCK IoStatus;
24452| BYTE PassedInOriginal;
24453| ULONG C0Offset=0;
24454|
24455| // DLOG((TEXT("%d files\n"),NumberOfFiles));
24456| for(i=0;i<NumberOfFiles;i++) {
24457|     PassedInOriginal=0;
24458|
24459| // DLOG((TEXT("%02d: '%S'\n"),i,Files[i]));
24460|
24461|     FileLen = wcslen(Files[i]);
24462|
24463|     FileNameOnly =
        | LocalAlloc(LPTR,(FileLen+257)*sizeof(WCHAR));
24464|
24465|     if(FileNameOnly) {
24466|         WCHAR *Colon = wcschr(Files[i],L':');
24467|
24468|         if(Colon) {
24469|             if(pGetVolumePathName) {
24470|                 wcsncpy(FileNameOnly,Files[i]);
24471|
                | if(pGetVolumePathName(Files[i],FileNameOnly,FileLen+257)
                | ) {
24472|                     /* it can be
24473|                     \\?e:\ !
24474|                     e:\ !
24475|                     */
24476|
                | if(FileNameOnly[(Colon-Files[i])+2] != L'0') {
24477|                     // must be a mount point..
                | we do not support yet
24478|                     DLOG((TEXT("File '%S' is on
                | mount point '%S'\n"),Files[i],FileNameOnly));
24479|                     LocalFree(FileNameOnly);
24480|                     return ERROR_INVALID_NAME;
24481|                 }
24482|             }
24483|         }
24484|
24485|
24486|         // get NT name
24487|
                | wcsncpy(Temp,Files[i],Colon-Files[i]+1);
24488|         Temp[Colon-Files[i]+1] = 0;
24489|
24490|         if(wcsncmp(Temp,L"\\\\"?\\",4)==0) {

```

```

24491|
    | memmove(Temp,Temp+4,(wcslen(Temp)-3)*sizeof(WCHAR));
24492|         }
24493|
24494|         if((Err =
    | GetNTDeviceName(Temp,VirtualDrive,256,TRUE))==0) {
24495|
24496| //             DLOG((TEXT("'%S' =
    | '%S'\n"),Temp,VirtualDrive));
24497|
24498|         | if(!PSMI_IsAnPSMVolume(VirtualDrive)) {
24499|             // okay, the user passed in
    | something like C:
24500|             // instead of the snapshot
    | drive M:
24501|
    | wcscpy(OriginalDrive,VirtualDrive);
24502|             PassedInOriginal = 1;
24503|
    | GetVirtualFromOriginal(SnapShot,OriginalDrive,VirtualDri
    | ve,256);
24504| //             DLOG((TEXT("PSM volume =
    | '%S'\n"),VirtualDrive));
24505|         }
24506|
24507|         Err =
    | GetVolumeClusterSize(hDevice,hEvent,VirtualDrive,&Volume
    | ClusterSize,&C0Offset);
24508| //             DLOG((TEXT("Cluster size = %08x,
    | c0=%08x, Err=%08x\n"),VolumeClusterSize,C0Offset,Err));
24509|
24510|         wcscpy(FileNameOnly,VirtualDrive);
24511|
24512| DoOpen:
24513|
    | if(FileNameOnly[wcslen(FileNameOnly)-1] != L'\\') {
24514|         wscat(FileNameOnly,L"\\");
24515|     }
24516|
24517|         wscat(FileNameOnly,Colon+2);
24518|
24519|         RtlInitUnicodeString( &Uni,
    | FileNameOnly );
24520|
24521|         InitializeObjectAttributes (
    | &ObjectAttributes,
24522|             &Uni,
24523|
    | OBJ_CASE_INSENSITIVE,

```

```

24524|                                     NULL,
24525|                                     NULL
    | );
24526|
24527|             Err = NtCreateFile( &FileHandle,
24528|             | FILE_WRITE_ATTRIBUTES |
24529|             | FILE_READ_ATTRIBUTES |
24530|             | SYNCHRONIZE,           // desired access
24531|             | &ObjectAttributes,     // object attributes
24532|                                     &IoStatus,
24533|                                     NULL,
    | // alloc size
24534|             | FILE_ATTRIBUTE_NORMAL, // file attributes
24535|             | FILE_SHARE_WRITE | FILE_SHARE_READ,
    | // share access
24536|                                     FILE_OPEN,
    | // create disposition
24537|             | FILE_NO_INTERMEDIATE_BUFFERING |
24538|             | FILE_SYNCHRONOUS_IO_NONALERT |
24539|             | FILE_OPEN_FOR_BACKUP_INTENT,           // create
    | options
24540|                                     NULL, //
    | eabuffer
24541|                                     0 ); //
    | ealength
24542|
24543|             if(!Err) {
24544|                 Err = NtQueryInformationFile(
24545|                 | FileHandle,
    | // IN HANDLE FileHandle,
24546|                 | &IoStatus,
    | // OUT PIO_STATUS_BLOCK IoStatusBlock,
24547|                 | &FSInfo,           //
    | IN PVOID FileInformation,
24548|                 | sizeof(FILE_STANDARD_INFORMATION), // IN ULONG
    | Length,
24549|                 | FileStandardInformation//
    | IN FILE_INFORMATION_CLASS FileInformationClass
24550|                 );
24551|                 Err = NtQueryInformationFile(

```

```

24552|             FileHandle,
| // IN HANDLE FileHandle,
24553|             &IoStatus,
| // OUT PIO_STATUS_BLOCK IoStatusBlock,
24554|             &FBInfo,          //
| IN PVOID FileInformation,
24555|
| sizeof(FILE_BASIC_INFORMATION),      // IN ULONG
| Length,
24556|             FileBasicInformation// IN
| FILE_INFORMATION_CLASS FileInformationClass
24557|         );
24558|
24559|         if(FSInfo.EndOfFile.QuadPart !=
| 0) {
24560|             // assume 1 extent
24561|             RPSize =
| sizeof(RETRIEVAL_POINTERS_BUFFER);
24562|             RP=LocalAlloc(LPTR,RPSize);
24563|             if(!RP) {
24564|                 Err =
| ERROR_OUTOFMEMORY;
24565|                 DLOG((TEXT("Error %08x
| allocating memory\n"),Err));
24566|                 goto SExit;
24567|             }
24568|             Err =
| FS_GetRetrievalPointers( FileHandle, &SVIB, RP, RPSize
| );
24569|             Count = 16;
24570|             while(
| (Err==STATUS_BUFFER_OVERFLOW) ) {
24571|                 // more than 1 extent,
| so calc how many we need
24572|                 RPSize =
| (Count*(2*sizeof(LARGE_INTEGER)))+sizeof(RETRIEVAL_POINT
| ERS_BUFFER)-(2*sizeof(LARGE_INTEGER));
24573|                 LocalFree(RP);
24574|
24575|                 RP =
| LocalAlloc(LPTR,RPSize);
24576|                 if(!RP) {
24577|                     Err =
| ERROR_OUTOFMEMORY;
24578|                     DLOG((TEXT("Error
| %08x allocating %d memory\n"),Err,RPSize));
24579|                     break;
24580|                 }
24581|                 Err =
| FS_GetRetrievalPointers( FileHandle, &SVIB, RP, RPSize

```

```

    | );
24582|             Count*=2;
24583|         }
24584|         // only display stuff
    | greater than 1 cluster (AKA a fragmented file)
24585|         // under ntfs it is
    | possible to have a file with 0 extents (its in the mft)
24586|         if(!Err) &&
    | (RP->ExtentCount>0)) {
24587|             // allocate enough
    | memory for LocalFree ranges
24588|
    | Ranges=LocalAlloc(LPTR,sizeof(tPSM_FreeRanges)+sizeof(tP
    | SM_RangeList)*RP->ExtentCount);
24589|             if(Ranges) {
24590|                 ULONG RangeCount=0;
24591|
24592|
    | wcscpy(Ranges->VolumeName,VirtualDrive);
24593|
    | Ranges->KernelSnapShotPointer =
    | SnapShot->KernelSnapShotPointer;
24594|
24595|                 StartVcn =
    | RP->StartingVcn;
24596|
    | for(j=0;j<RP->ExtentCount;j++) {
24597| //
    | DLOG((TEXT("%04d %04d StartVcn  = %l64d
    | LCN=%l64d\n"),j,RangeCount,StartVcn.QuadPart,RP->Extents
    | [j].Lcn.QuadPart));
24598| //
    | DLOG((TEXT("
    | Next      = %l64d
    | (%l64d\n"),RP->Extents[j].NextVcn.QuadPart,RP->Extents[
    | j].NextVcn.QuadPart-StartVcn.QuadPart));
24599|         // skip over if
    | no entry allocated (file is compressed or sparse)
24600|
    | if(RP->Extents[j].Lcn.QuadPart!=-1) {
24601|
24602|             // NT 3.51
    | doesnt support __allmul so we cant just do
    | .QuadPart*.QuadPart
24603|             LI =
    | RtlExtendedIntegerMultiply(RP->Extents[j].Lcn,VolumeClus
    | terSize);
24604|
    | Ranges->RangeList[RangeCount].Offset.QuadPart =
    | LI.QuadPart+C0Offset;
24605|             LI.QuadPart

```

```

    | = RP->Extents[j].NextVcn.QuadPart-StartVcn.QuadPart;
24606|          LI =
    | RtlExtendedIntegerMultiply(LI,VolumeClusterSize);
24607|
    | Ranges->RangeList[RangeCount].Count =
    | (ULONG)LI.LowPart;
24608| //
    | DLOG((TEXT("%I64x-%I64x
    | (%I64x)\n"),Ranges->RangeList[RangeCount].Offset.QuadPar
    | t,Ranges->RangeList[RangeCount].Offset.QuadPart+Ranges->
    | RangeList[RangeCount].Count,Ranges->RangeList[RangeCount
    | ].Count));
24609|
24610|          StartVcn =
    | RP->Extents[j].NextVcn;
24611|
    | RangeCount++;
24612|          }
24613|          }
24614|
24615|
    | Ranges->NumberOfRanges = RangeCount;
24616|          Err =
    | PSMI_FreeRanges(hDevice,hEvent,Ranges);
24617|
24618|          LocalFree(Ranges);
24619|          } else {
24620|          Err =
    | ERROR_OUTOFMEMORY;
24621|          DLOG((TEXT("Error
    | %08x allocating memory\n"),Err));
24622|          }
24623|          } else {
24624|          // need to free mft
    | entry if ntfs!
24625|          //DLOG((TEXT("File has
    | no extents\n")));
24626|          }
24627|          LocalFree(RP);
24628|          SExit:
24629|          if(PassedInOriginal==2) {
24630|          // set times back as
    | its on the original drive
24631|          Err =
    | NtSetInformationFile(
24632|          FileHandle,
    | // IN HANDLE FileHandle,
24633|          &IoStatus,
    | // OUT PIO_STATUS_BLOCK IoStatusBlock,
24634|          &FBInfo,

```



```

    | // IN PVOID FileInformation,
24635|
    | sizeof(FILE_BASIC_INFORMATION),      // IN ULONG
    | Length,
24636|
    | FileBasicInformation// IN FILE_INFORMATION_CLASS
    | FileInformationClass
24637|         );
24638|         if(Err) {
24639|             DLOG((TEXT("Error
    | %08x setting timestamps back\n"),Err));
24640|             Err = 0;
24641|         }
24642|     }
24643|     NtClose(FileHandle);
24644|
24645|     } else {
24646|         //DLOG((TEXT("File is zero
    | length\n")));
24647|         Err = 0;
24648|     }
24649|     } else {
24650|         DLOG((TEXT("Error %08x opening
    | file '%S'\n"),Err,FileNameOnly));
24651|         if(PassedInOriginal==1) {
24652|             // so we dont do it again
24653|             PassedInOriginal= 2;
24654|
    | wcsncpy(FileNameOnly,OriginalDrive);
24655|             DLOG((TEXT("Opening file on
    | '%S' instead\n"),OriginalDrive));
24656|             goto DoOpen;
24657|         }
24658|     }
24659|     } // if GetNTDeviceName
24660|     } // if Colon
24661|     LocalFree(FileNameOnly);
24662| } else { // if(FileNameOnly)
24663|     Err = ERROR_OUTOFMEMORY;
24664| }
24665|
24666| // we do not look at error, and continue on the
    | next file in the list
24667|     if(!SaveErr) {
24668|         SaveErr = Err;
24669|     }
24670| } // for i
24671| //DLOG((TEXT("FreeFiles returning
    | %08x\n"),SaveErr));
24672| return SaveErr;

```

```

24673| }
24674|
24675| STATIC ULONG PSMI_FreeVolume ( HANDLE hDevice, HANDLE
    | hEvent, tPSM_FreeVolume *Volume)
24676| {
24677|     BOOL B=FALSE;
24678|     ULONG Err=0;
24679|     ULONG returned=0;
24680|     OVERLAPPED o;
24681|
24682|     if(hDevice) {
24683|         memset(&o,0,sizeof(o));
24684|         ResetEvent(hEvent);
24685|         o.hEvent = hEvent;
24686|
24687|         B = DeviceIoControl( hDevice,
24688|             IOCTL_FREE_VOLUME,
24689|             Volume,
24690|             sizeof(tPSM_FreeVolume),
24691|             NULL,
24692|             0,
24693|             &returned,
24694|             &o);
24695|
24696|         if (B) {
24697|             Err = 0;
24698|         } else {
24699|             Err = GetLastError();
24700|             if (Err == ERROR_IO_PENDING) {
24701|                 WaitForSingleObject ( hEvent, INFINITE
    | );
24702|                 Err = GetLastError();
24703|             }
24704|         }
24705|     } else {
24706|         Err = ERROR_INVALID_HANDLE;
24707|     }
24708|
24709|     return Err;
24710|
24711| }
24712|
24713|
24714| STATIC ULONG PSMI_GetError ( HANDLE hDevice, HANDLE
    | hEvent, pPSM_GetErrorOut Error, pSnapShot SnapShot)
24715| {
24716|     BOOL B=FALSE;
24717|     ULONG Err=0;
24718|     ULONG returned=0;
24719|     OVERLAPPED o;

```

```

24720|
24721|     if(hDevice) {
24722|         memset(&o,0,sizeof(o));
24723|         ResetEvent(hEvent);
24724|         o.hEvent = hEvent;
24725|
24726|         B = DeviceIoControl( hDevice,
24727|             IOCTL_GET_ERROR,
24728|             SnapShot ?
                | &SnapShot->KernelSnapShotPointer : NULL,
24729|             SnapShot ?
                | sizeof(SnapShot->KernelSnapShotPointer) : 0,
24730|             Error,
24731|             sizeof(tPSM_GetErrorOut),
24732|             &returned,
24733|             &o);
24734|
24735|         if (B) {
24736|             Err = 0;
24737|         } else {
24738|             Err = GetLastError();
24739|             if (Err == ERROR_IO_PENDING) {
24740|                 WaitForSingleObject ( hEvent, INFINITE
                | );
24741|                 Err = GetLastError();
24742|             }
24743|         }
24744|     } else {
24745|         Err = ERROR_INVALID_HANDLE;
24746|     }
24747|
24748|     return Err;
24749|
24750| }
24751|
24752| STATIC ULONG PSMI_GetProgress ( HANDLE hDevice, HANDLE
    | hEvent, pPSM_GetProgressOut Progress, pSnapShot
    | SnapShot)
24753| {
24754|     BOOL B=FALSE;
24755|     ULONG Err=0;
24756|     ULONG returned=0;
24757|     OVERLAPPED o;
24758|
24759|     if(hDevice) {
24760|         memset(&o,0,sizeof(o));
24761|         ResetEvent(hEvent);
24762|         o.hEvent = hEvent;
24763|
24764|         B = DeviceIoControl( hDevice,

```

```

24765|         IOCTL_GET_PROGRESS,
24766|         SnapShot ?
    | &SnapShot->KernelSnapShotPointer : NULL,
24767|         SnapShot ?
    | sizeof(SnapShot->KernelSnapShotPointer) : 0,
24768|         Progress,
24769|         sizeof(tPSM_GetProgressOut),
24770|         &returned,
24771|         &o);
24772|
24773|     if (B) {
24774|         Err = 0;
24775|     } else {
24776|         Err = GetLastError();
24777|         if (Err == ERROR_IO_PENDING) {
24778|             WaitForSingleObject ( hEvent, INFINITE
    | );
24779|             Err = GetLastError();
24780|         }
24781|     }
24782| } else {
24783|     Err = ERROR_INVALID_HANDLE;
24784| }
24785|
24786| return Err;
24787|
24788| }
24789|
24790| STATIC ULONG PSMI_OpenExclusive ( HANDLE hDevice,
    | HANDLE hEvent, pSnapShot SnapShot )
24791| {
24792|     BOOL B=FALSE;
24793|     ULONG Err=0;
24794|     ULONG returned=0;
24795|     OVERLAPPED o;
24796|
24797|     if(hDevice) {
24798|         memset(&o,0,sizeof(o));
24799|         ResetEvent(hEvent);
24800|         o.hEvent = hEvent;
24801|
24802|         B = DeviceIoControl( hDevice,
24803|             IOCTL_OPEN_EX,
24804|             &SnapShot->KernelSnapShotPointer,
24805|             | sizeof(SnapShot->KernelSnapShotPointer),
24806|             NULL,
24807|             0,
24808|             &returned,
24809|             &o);

```

```

24810|
24811|     if (B) {
24812|         Err = 0;
24813|     } else {
24814|         Err = GetLastError();
24815|         if (Err == ERROR_IO_PENDING) {
24816|             WaitForSingleObject ( hEvent, INFINITE
24817| | );
24818|             Err = GetLastError();
24819|         }
24820|     } else {
24821|         Err = ERROR_INVALID_HANDLE;
24822|     }
24823|
24824|     return Err;
24825|
24826| }
24827|
24828| STATIC ULONG PSMI_CloseExclusive ( HANDLE hDevice,
24829| | HANDLE hEvent, pSnapShot SnapShot )
24830| {
24831|     BOOL B=FALSE;
24832|     ULONG Err=0;
24833|     ULONG returned=0;
24834|     OVERLAPPED o;
24835|     if(hDevice) {
24836|         memset(&o,0,sizeof(o));
24837|         ResetEvent(hEvent);
24838|         o.hEvent = hEvent;
24839|
24840|         B = DeviceIoControl( hDevice,
24841| |         IOCTL_CLOSE_EX,
24842| |         &SnapShot->KernelSnapShotPointer,
24843| |         sizeof(SnapShot->KernelSnapShotPointer),
24844| |         NULL,
24845| |         0,
24846| |         &returned,
24847| |         &o);
24848|
24849|         if (B) {
24850|             Err = 0;
24851|         } else {
24852|             Err = GetLastError();
24853|             if (Err == ERROR_IO_PENDING) {
24854|                 WaitForSingleObject ( hEvent, INFINITE
24855| | );
24856|                 Err = GetLastError();

```

```

24856|     }
24857| }
24858| } else {
24859|     Err = ERROR_INVALID_HANDLE;
24860| }
24861|
24862| return Err;
24863|
24864| }
24865|
24866|
24867|
24868| STATIC ULONG PSMI_Close_Internal( HANDLE hDevice,
    | HANDLE hEvent, PVOID KernelSnapShot )
24869| {
24870|     BOOL B=FALSE;
24871|     ULONG Err=0;
24872|     ULONG returned=0;
24873|     OVERLAPPED o;
24874|     tClosePSMInternal Out;
24875|     pThreadStorage ThreadStorage = GetThreadStorage();
24876|
24877|     if(hDevice) {
24878|         memset(&o,0,sizeof(o));
24879|         ResetEvent(hEvent);
24880|         o.hEvent = hEvent;
24881|         Out.KernelSnapShotPointer = KernelSnapShot;
24882|
24883|         B = DeviceIoControl( hDevice,
24884|             IOCTL_TURNOFF_PSM,
24885|             &Out,
24886|             sizeof(Out),
24887|             NULL,
24888|             0,
24889|             &returned,
24890|             &o);
24891|
24892|         if (B) {
24893|             Err = 0;
24894|         } else {
24895|             Err = GetLastError();
24896|             if (Err == ERROR_IO_PENDING) {
24897|                 WaitForSingleObject ( hEvent, INFINITE
                | );
24898|                 Err = GetLastError();
24899|             }
24900|         }
24901|     } else {
24902|         Err = ERROR_INVALID_HANDLE;
24903|     }

```

```

24904|
24905| // no longer valid after call to close.
24906| ThreadStorage->AbortEvent = NULL;
24907|
24908| return Err;
24909| }
24910|
24911| PSMSTATUS PSMAPI Psmi_GetKernelSnapShotVolumesW (
24912|     PVOID KernelSnapShotPointer,
24913|     WCHAR *Buffer,
24914|     ULONG BufferSize )
24915| {
24916|     ULONG B=FALSE;
24917|     LONG Err=0;
24918|     OVERLAPPED o;
24919|     ULONG returned=0;
24920|     pThreadStorage ThreadStorage = GetThreadStorage();
24921|     tPSM_GetKernelSnapShotInfoIn In;
24922|     WCHAR *TempBuffer=LocalAlloc(LPTR,128*1024);
24923|     ULONG BytesLeft=BufferSize;
24924|
24925|     if(!TempBuffer) {
24926|         return ERROR_OUTOFMEMORY;
24927|     }
24928|
24929|     __try {
24930|         memset(Buffer,0,BufferSize);
24931|
24932|         if
            | ((ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) ) {
24933|             memset(&o,0,sizeof(o));
24934|             o.hEvent = CreateEvent( NULL, FALSE, FALSE,
            | NULL );
24935|
24936|             In.KernelSnapShotPointer =
            | KernelSnapShotPointer;
24937|
24938|             B = DeviceIoControl(
            | ThreadStorage->PSManHandle,
24939|             IOCTL_GET_KERNEL_SNAPSHOT_VOLUMES,
24940|             &In,
24941|             sizeof(In),
24942|             TempBuffer,
24943|             128*1024,
24944|             &returned,
24945|             &o);
24946|
24947|             if(B) {
24948|                 WCHAR *p = TempBuffer;
24949|                 Err = 0;

```

```

24950|         while(*p) {
24951|             ULONG LenChars = wcslen(p) + 1;
24952|             ULONG LenBytes = LenChars *
| sizeof(WCHAR);
24953|             if ( BytesLeft < LenBytes ) {
24954|                 Err = ERROR_MORE_DATA;
24955|                 break;
24956|             }
24957|             wcsncpy(Buffer,p);
24958|             Buffer += LenChars;
24959|             p += LenChars;
24960|             BytesLeft -= LenBytes;
24961|         }
24962|         if ( !Err ) {
24963|             if ( BytesLeft > 0 ) {
24964|                 *Buffer = L'\0'; // 2 null
| terminators in a row indicate end-of-list
24965|             } else {
24966|                 Err = ERROR_MORE_DATA;
24967|             }
24968|         }
24969|     } else {
24970|         Err = GetLastError();
24971|         DLOG((TEXT("Error %08x Calling
| Psm_GetKernelSnapShotVolumes\n"),Err));
24972|     }
24973|     CloseHandle(o.hEvent);
24974| } else {
24975|     Err = ERROR_INVALID_HANDLE;
24976| }
24977| LocalFree(TempBuffer);
24978| } __except(EXCEPTION_EXECUTE_HANDLER) {
24979|     Err = GetExceptionCode();
24980|     DLOG((TEXT("Exception %08x
| Psm_GetKernelSnapShotVolumes\n"),Err));
24981| }
24982|
24983| return Err;
24984|
24985| }
24986|
24987| STATIC ULONG PSMI_Close( HANDLE hDevice, HANDLE hEvent,
| tSnapShot *SnapShot )
24988| {
24989|     BOOL B=FALSE;
24990|     ULONG Err=0;
24991|     ULONG returned=0;
24992|     pthread_storage ThreadStorage = GetThreadStorage();
24993|     ULONG i;
24994|

```



```

24995|  if(Psm_IsPersistentSnapShotPointer(SnapShot)) {
24996|      WCHAR Name[256];
24997|      WCHAR *Buffer;
24998|      ULONG BufferSize=128*1024;
24999|
25000|      Buffer = LocalAlloc(LPTR,BufferSize);
25001|      if(Buffer) {
25002|          Err = Psmi_GetKernelSnapShotVolumesW(
25003|              SnapShot,
25004|              Buffer,
25005|              BufferSize);
25006|
25007|          if(!Err) {
25008|              WCHAR *p=Buffer;
25009|              WCHAR Temp[256];
25010|
25011|              while(*p) {
25012|                  wcscpy(Temp,L "");
25013|
25014|                  Err =
25015|                      | GetDriveLetterForNtDeviceName(p, Temp);
25016|                      if(Err) {
25017|                          Err =
25018|                              | GetWin32NameForNtDeviceName(p,Temp);
25019|                              }
25020|                              #if 0 //removed this remove since this is for
25021|                                  | persistent snapshots only and they are now removed in
25022|                                  | kernel mode
25023|                                  // NOTE !! this code was dismounting the LIVE
25024|                                  | volume by mistake so you'll need to correct that before
25025|                                  | you
25026|                                  // should ever think of reinstating it
25027|                                  | !!!!!!!!!
25028|                                  // dismount volume
25029|                                  DLOG((TEXT("dismounting volume
25030|                                  | mount point '%S'\n"),p));
25031|                                  RemoveDeviceName(p);
25032|                              #endif
25033|                              // remove junction point
25034|                              wcscpy(Name,Temp);
25035|                              wscat(Name,L "\\");
25036|
25037|                              // we should free drive letters
25038|                              | here, but we have no idea what the mapping is
25039|                              // so until we do, we will just let
25040|                              | the kernel mode driver cleanup the drive
25041|                              // letters
25042|                              DLOG((TEXT("Getting directory name
25043|                              | for '%S'\n"),Name));

```

```

25034|
    | Psm_GetUserName(SnapShot,&Name[wcslen(Name)],(256-wcslen
    | (Name))*sizeof(WCHAR));
25035|         DLOG((TEXT("removing volume mount
    | point '%S'\n"),Name));
25036|         DeleteJunction(Name);
25037|         p+=wcslen(p)+1;
25038|     }
25039|     Err =
    | PSMI_Close_Internal(hDevice,hEvent,SnapShot);
25040|     }
25041|     LocalFree(Buffer);
25042| } else {
25043|     Err= ERROR_OUTOFMEMORY;
25044| }
25045| return Err;
25046| }
25047|
25048| // fixfixfix what to do if invalid snapshot is
    | passed in?
25049| // do we close last snapshot or do we return error.
25050| // problem here is that nobody every checks the
    | return values
25051| // on closes so if they are a service they will
    | lose memory and
25052| // not know why.
25053| // NULL is a valid parameter!
25054|
25055| if((SnapShot) && (IsValidSnapShot(SnapShot))) {
25056|     DLOG(("PSMI_Close called with snapshot %08x %d
    | (%08x)\n",SnapShot,SnapShot->Instance,SnapShot->KernelSn
    | apShotPointer));
25057|     WaitForSingleObject ( SharedMemoryMutex,
    | INFINITE );
25058|     __try {
25059|         FreeVolumesForSnapShot(SnapShot);
25060|     } __finally {
25061|         ReleaseMutex(SharedMemoryMutex);
25062|     }
25063|     Err =
    | PSMI_Close_Internal(hDevice,hEvent,SnapShot->KernelSnapS
    | hotPointer);
25064| } else {
25065|     if(ThreadStorage->NumSnapShots>0) {
25066|         PVOID SnapShotSave = SnapShot;
25067|         SnapShot =
    | MakePointer(ThreadStorage->SnapShotsOffset[ThreadStorage
    | ->NumSnapShots-1]);
25068|         DLOG(("PSMI_Close called with %08x, using
    | snapshot %08x %d

```

```

    | (%08x)\n",SnapShotSave,SnapShot,SnapShot->Instance,SnapS
    | hot->KernelSnapShotPointer));
25069|         WaitForSingleObject ( SharedMemoryMutex,
    | INFINITE );
25070|         __try {
25071|             FreeVolumesForSnapShot(SnapShot);
25072|         } __finally {
25073|             ReleaseMutex(SharedMemoryMutex);
25074|         }
25075|         Err =
    | PSMI_Close_Internal(hDevice,hEvent,SnapShot->KernelSnapS
    | hotPointer);
25076|     } else {
25077|         DLOG(("PSMI_Close called with no
    | snapshots!!!\n"));
25078|         return ERROR_INVALID_PARAMETER;
25079|     }
25080| }
25081|
25082| // get rid of the temp file we created.
25083| if(ThreadStorage->UsedTempFile) {
25084| /* we know do this in kernel mode with
    | FILE_DELETE_ON_CLOSE
25085|     DLOG((TEXT("Deleting cache file\n")));
25086|     DeleteFileW(ThreadStorage->TempFileName);
25087| */
25088|     ThreadStorage->UsedTempFile = FALSE;
25089| }
25090|
25091| if(1){
25092|     WCHAR Str[30];
25093|     ULONG PsmErr;
25094|     tPSM_GetErrorOut PsmErrOut;
25095|
25096|     // get reason for psm closing.
25097|     PsmErr = Psm_GetError( &PsmErrOut );
25098|
25099|     swprintf(Str,L"%08x
    | %08x",PsmErr,PsmErrOut.ErrorCode);
25100|     DLOG((TEXT("Psm exited because of win32 error =
    | %08x, NT Error = %08x\n"),PsmErr,PsmErrOut.ErrorCode));
25101|
25102|     // execute postopen file
25103|
    | ExecuteFile(PostCloseFile,Str,ChildUser,ChildPassword);
25104| }
25105|
25106| // no longer valid after call to close.
25107| ThreadStorage->AbortEvent = NULL;
25108|

```

```

25109|   i=0;
25110|   while(i<ThreadStorage->NumSnapShots) {
25111|       | if(MakePointer(ThreadStorage->SnapShotsOffset[i]) ==
        | SnapShot) {
25112|           | memmove(&ThreadStorage->SnapShotsOffset[i],&ThreadStorag
        | e->SnapShotsOffset[i+1],(ThreadStorage->NumSnapShots-i-1
        | )*sizeof(ULONG));
25113|           ThreadStorage->NumSnapShots--;
25114|       } else {
25115|           i++;
25116|       }
25117|   }
25118|
25119|
25120|   WaitForSingleObject ( SharedMemoryMutex, INFINITE
        | );
25121|   __try {
25122|       i=0;
25123|       while(i<SharedMemory->NumberOfSnapShots) {
25124|           | if(MakePointer(SharedMemory->SnapShotsOffset[i]) ==
        | SnapShot) {
25125|               | memmove(&SharedMemory->SnapShotsOffset[i],&SharedMemory-
        | >SnapShotsOffset[i+1],(SharedMemory->NumberOfSnapShots-i
        | -1)*sizeof(ULONG));
25126|               SharedMemory->NumberOfSnapShots--;
25127|           } else {
25128|               i++;
25129|           }
25130|       }
25131|   } __finally {
25132|       ReleaseMutex(SharedMemoryMutex);
25133|   }
25134|
25135|   FreeSnapShot(SnapShot);
25136|
25137|   return Err;
25138| }
25139|
25140| STATIC ULONG PSMI_CloseManager( HANDLE hDevice )
25141| {
25142|   ULONG Err=0;
25143|
25144|   if(hDevice) {
25145|       CloseHandle(hDevice);
25146|       Err = 0;
25147|   } else {

```

```

25148|     Err = ERROR_INVALID_HANDLE;
25149| }
25150|
25151| return Err;
25152| }
25153|
25154| //
25155| // Undocumented FSCTL_SET_REPARSE_POINT structure
    | definition
25156| //
25157| #define REPARSE_MOUNTPOINT_HEADER_SIZE 8
25158| typedef struct {
25159|     DWORD      ReparseTag;
25160|     DWORD      ReparseDataLength;
25161|     WORD       Reserved;
25162|     WORD       ReparseTargetLength;
25163|     WORD       ReparseTargetMaximumLength;
25164|     WORD       Reserved1;
25165|     WCHAR      ReparseTarget[1];
25166| } REPARSE_MOUNTPOINT_DATA_BUFFER,
    | *PREPARSE_MOUNTPOINT_DATA_BUFFER;
25167|
25168| /*
25169| int CreateJunction( PWCHAR LinkDirectory, PWCHAR
    | LinkTarget, PLARGE_INTEGER Time )
25170| {
25171|     char      reparseBuffer[MAX_PATH*3];
25172|     WCHAR      targetNativeFileName[MAX_PATH];
25173|     HANDLE     hFile = INVALID_HANDLE_VALUE;
25174|     PREPARSE_MOUNTPOINT_DATA_BUFFER reparseInfo =
25175|         (PREPARSE_MOUNTPOINT_DATA_BUFFER)
    | reparseBuffer;
25176|     UNICODE_STRING UniName={0};
25177|     OBJECT_ATTRIBUTES ObjectAttributes={0};
25178|     IO_STATUS_BLOCK IoStatus;
25179|     NTSTATUS Status;
25180|
25181|     DLOG((TEXT("Creating junction for '%S' to
    | '%S'\n"),LinkDirectory,LinkTarget));
25182|
25183|     wcsncpy(targetNativeFileName,LinkTarget);
25184|     RtlInitUnicodeString( &UniName, LinkDirectory );
25185|
25186|     if( targetNativeFileName[wcslen(
    | targetNativeFileName )-1] != L'\\' ) {
25187|         wscat( targetNativeFileName, L"\\");
25188|     }
25189|
25190|
25191|     //

```

```

25192| // Create the link
25193| //
25194| InitializeObjectAttributes ( &ObjectAttributes,
25195|                             &UniName,
25196|                             OBJ_CASE_INSENSITIVE,
25197|                             NULL,
25198|                             NULL );
25199|
25200| Status = NtCreateFile( &hFile,
25201|                       GENERIC_WRITE,
25202|                       | // desired access
25203|                       &ObjectAttributes,
25204|                       | // object attributes
25205|                       &IoStatus,
25206|                       NULL, //
25207|                       | alloc size
25208|                       FILE_ATTRIBUTE_NORMAL,
25209|                       | // file attributes
25210|                       0,
25211|                       | // share access
25212|                       FILE_OPEN_IF,
25213|                       | // create disposition
25214|                       FILE_DIRECTORY_FILE|
25215|                       | FILE_OPEN_REPARSE_POINT|
25216|                       | FILE_OPEN_FOR_BACKUP_INTENT, // create
25217|                       | options
25218|                       NULL, // eabuffer
25219|                       0 ); // ealength
25220|
25221| if(!Status) {
25222|     // set time of directory
25223|     SetFileTime( hFile, (FILETIME*)Time,
25224|                 | (FILETIME*)Time, (FILETIME*)Time);
25225|
25226|     // Build the reparse info
25227|     memset( reparseInfo, 0, sizeof( *reparseInfo
25228|     | ));
25229|     reparseInfo->ReparseTag =
25230|     | IO_REPARSE_TAG_MOUNT_POINT;
25231|
25232|     reparseInfo->ReparseTargetLength = wcslen(
25233|     | targetNativeFileName ) * sizeof(WCHAR);
25234|     reparseInfo->ReparseTargetMaximumLength =
25235|     | reparseInfo->ReparseTargetLength + sizeof(WCHAR);
25236|     wcscpy( reparseInfo->ReparseTarget,
25237|     | targetNativeFileName );
25238|     reparseInfo->ReparseDataLength =

```

```

    | reparseInfo->ReparseTargetLength + 12;
25227|
25228|     // Set the link
25229|     Status = NtDeviceIoControlFile(
25230|         hFile,
25231|         NULL,
25232|         NULL,
25233|         NULL,
25234|         &IoStatus,
25235|         FSCTL_SET_REPARSE_POINT,
25236|         reparseInfo,
25237|         reparseInfo->ReparseDataLength +
    | REPARSE_MOUNTPOINT_HEADER_SIZE,
25238|         NULL,
25239|         0
25240|     );
25241|
25242|     if(Status==0) {
25243|         DLOG((TEXT("Success creating
    | junction\n")));
25244|     } else {
25245|         FILE_DISPOSITION_INFORMATION Del={0};
25246|
25247|         DLOG((TEXT("Error %08x creating
    | junction\n"),Status));
25248|         // delete file if we could not create it
25249|         Del.DeleteFile = TRUE;
25250|         NtSetInformationFile( hFile,
25251|             &IoStatus, &Del, sizeof(Del),
25252|             FileDispositionInformation
25253|         );
25254|     }
25255| } else {
25256|     DLOG((TEXT("Error %08x opening
    | directory\n"),Status));
25257| }
25258|
25259| if ( hFile != INVALID_HANDLE_VALUE ) {
25260|     NtClose(hFile);
25261| }
25262| return Status;
25263| }
25264| */
25265|
25266| // Older SDK's had the wrong value specified for this
    | ioctl, so lets
25267| // define it correctly
25268| #ifndef FILE_SPECIAL_ACCESS
25269|     #define FILE_SPECIAL_ACCESS (FILE_ANY_ACCESS)
25270| #undef FSCTL_SET_REPARSE_POINT

```

```

25271|  #define FSCTL_SET_REPARSE_POINT
      | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 41, METHOD_BUFFERED,
      | FILE_SPECIAL_ACCESS) // REPARSE_DATA_BUFFER,
25272| #endif
25273|
25274| //-----
      | -----
25275| //
25276| // CreateJunction
25277| //
25278| // This routine creates a NTFS junction, using the
      | undocumented
25279| // FSCTL_SET_REPARSE_POINT structure Win2K uses for
      | mount points
25280| // and junctions.
25281| //
25282| //-----
      | -----
25283| int CreateJunction2( PWCHAR LinkDirectory, PWCHAR
      | LinkTarget, PLARGE_INTEGER Time )
25284| {
25285|     NTSTATUS    status = 0;
25286|     char        reparseBuffer[MAX_PATH*3];
25287|     WCHAR        directoryFileName[MAX_PATH];
25288|     WCHAR        targetFileName[MAX_PATH];
25289|     WCHAR        targetNativeFileName[MAX_PATH];
25290|     PWCHAR        filePart;
25291|     HANDLE        hFile = INVALID_HANDLE_VALUE;
25292|     DWORD        returnedLength;
25293|     PREPARSE_MOUNTPOINT_DATA_BUFFER reparseInfo =
25294|         (PREPARSE_MOUNTPOINT_DATA_BUFFER)
      | reparseBuffer;
25295|
25296| #if 0
25297|     //
25298|     // Get the full path referenced by the target
25299|     //
25300|     if( !GetFullPathName( LinkTarget,
25301|                           MAX_PATH, targetFileName,
25302|                           &filePart )) {
25303|
25304|         _tprintf(_T("%s is an invalid file name:\n"),
      | LinkTarget );
25305|         PrintWin32Error( GetLastError());
25306|         return -1;
25307|     }
25308| #else
25309|     wcscpy(targetFileName,LinkTarget);
25310| #endif
25311|

```



```

25312|
25313| //
25314| // Get the full path referenced by the directory
25315| //
25316| if( !GetFullPathNameW( LinkDirectory,
25317|                      MAX_PATH, directoryFileName,
25318|                      &filePart )) {
25319|
25320|     DLOG((TEXT("%s is an invalid file name:
25321| | %08x\n"), LinkDirectory,GetLastError()) );
25322|     return GetLastError();
25323| }
25324| //
25325| // Make the native target name
25326| //
25327| // _stprintf( targetNativeFileName, _T("\\??\\%s"),
25328| | targetFileName );
25329| wscpy(targetNativeFileName ,targetFileName);
25330| if( targetNativeFileName[wcslen(
25331| | targetNativeFileName )-1] != L'\\') {
25332|     wscat( targetNativeFileName, L"\\");
25333| }
25334| //
25335| // Create the link - ignore errors since it might
25336| | already exist
25337| //
25338| CreateDirectoryW( LinkDirectory, NULL );
25339| Login_EnablePrivilege ( SE_BACKUP_NAME );
25340| // Login_EnablePrivilege ( SE_TCB_NAME );
25341| // Login_EnablePrivilege ( SE_RESTORE_NAME );
25342| // Login_EnablePrivilege ( SE_CREATE_PERMANENT_NAME );
25343|
25344| hFile = CreateFileW( LinkDirectory, GENERIC_WRITE,
25345| | 0,
25346| | NULL, OPEN_EXISTING,
25347| | FILE_FLAG_OPEN_REPARSE_POINT|FILE_FLAG_BACKUP_SEMANTICS,
25348| | NULL );
25349| if( hFile == INVALID_HANDLE_VALUE ) {
25350|     DLOG((TEXT("Error %08x creating %s:\n"),
25351| | GetLastError(),LinkDirectory ));
25352|     return GetLastError();
25353| }
25354| // set time of directory

```

```

25354|   SetFileTime( hFile, (FILETIME*)Time,
      | (FILETIME*)Time, (FILETIME*)Time);
25355|
25356|   //
25357|   // Build the reparse info
25358|   //
25359|   memset( reparseInfo, 0, sizeof( *reparseInfo ));
25360|   reparseInfo->ReparseTag =
      | IO_REPARSE_TAG_MOUNT_POINT;
25361|
25362|   reparseInfo->ReparseTargetLength = wcslen(
      | targetNativeFileName ) * sizeof(WCHAR);
25363|   reparseInfo->ReparseTargetMaximumLength =
      | reparseInfo->ReparseTargetLength + sizeof(WCHAR);
25364|   wcscpy( reparseInfo->ReparseTarget,
      | targetNativeFileName );
25365|   reparseInfo->ReparseDataLength =
      | reparseInfo->ReparseTargetLength + 12;
25366|
25367|   DLOG((TEXT("rt=%08x, rdl=%08x, r=%04x, rtl=%04x,
      | rtml=%04x, r1=%04x, '%S'\n"),
25368|         reparseInfo->ReparseTag,
25369|         reparseInfo->ReparseDataLength,
25370|         reparseInfo->Reserved,
25371|         reparseInfo->ReparseTargetLength,
25372|         reparseInfo->ReparseTargetMaximumLength,
25373|         reparseInfo->Reserved1,
25374|         reparseInfo->ReparseTarget
25375|         ));
25376|
25377|   //
25378|   // Set the link
25379|   //
25380|   Login_EnablePrivilege ( SE_BACKUP_NAME );
25381| // Login_EnablePrivilege ( SE_TCB_NAME );
25382| // Login_EnablePrivilege ( SE_RESTORE_NAME );
25383| // Login_EnablePrivilege ( SE_CREATE_PERMANENT_NAME );
25384|   if( !DeviceIoControl( hFile,
      | FSCTL_SET_REPARSE_POINT,
25385|         reparseInfo,
25386|         reparseInfo->ReparseDataLength +
      | REPARSE_MOUNTPOINT_HEADER_SIZE,
25387|         NULL, 0, &returnedLength, NULL )) {
25388|
25389|       status = GetLastError();
25390|       DLOG((TEXT("Error %08x setting junction for
      | %S:\n"), status, LinkDirectory ));
25391|       CloseHandle( hFile );
25392|       RemoveDirectoryW( LinkDirectory );
25393|       return status;

```

```

25394|    }
25395|
25396|    CloseHandle (hFile);
25397|    DLOG((TEXT("Created: %S\nTargetted at:
    | %S\n"),directoryFileName, targetFileName ));
25398|    return status;
25399| }
25400|
25401|
25402| int DeleteJunction( PWCHAR Junction )
25403| {
25404|    Login_EnablePrivilege ( SE_BACKUP_NAME );
25405|    //Login_EnablePrivilege ( SE_TCB_NAME );
25406|    Login_EnablePrivilege ( SE_RESTORE_NAME );
25407|    //Login_EnablePrivilege ( SE_CREATE_PERMANENT_NAME
    | );
25408|
25409|    if(RemoveDirectoryW( Junction )) {
25410|        return 0;
25411|    } else {
25412|        DLOG((TEXT("Error %08x removing junction
    | '%S\n"),GetLastError(),Junction));
25413|        return GetLastError();
25414|    }
25415| }
25416|
25417| STATIC int SetTimeForFile( PWCHAR File, PLARGE_INTEGER
    | Time )
25418| {
25419|    WCHAR    targetNativeFileName[MAX_PATH];
25420|    HANDLE    hFile;
25421|    UNICODE_STRING    UniName={0};
25422|    OBJECT_ATTRIBUTES    ObjectAttributes={0};
25423|    IO_STATUS_BLOCK IoStatus;
25424|    NTSTATUS Status;
25425|
25426|    DLOG((TEXT("Setting time for '%S\n"),File));
25427|
25428|    if(File[1]==':') {
25429|        // convert to Object name space name
25430|        wcscpy(targetNativeFileName,L"\\??\\");
25431|        wcscat(targetNativeFileName,File);
25432|    } else {
25433|        // already a name space name
25434|        wcscpy(targetNativeFileName,File);
25435|    }
25436|
25437|    if( targetNativeFileName[wcslen(
    | targetNativeFileName )-1] != L'\\') {
25438|        wcscat( targetNativeFileName, L"\\");

```

```

25439| }
25440|
25441| RtlInitUnicodeString( &UniName,targetNativeFileName
| );
25442|
25443| InitializeObjectAttributes ( &ObjectAttributes,
25444|                             &UniName,
25445|                             OBJ_CASE_INSENSITIVE,
25446|                             NULL,
25447|                             NULL );
25448|
25449| Status = NtCreateFile( &hFile,
25450|                       GENERIC_WRITE,
| // desired access
25451|                       &ObjectAttributes,
| // object attributes
25452|                       &IoStatus,
25453|                       NULL,          //
| alloc size
25454|                       FILE_ATTRIBUTE_NORMAL,
| // file attributes
25455|                       0,
| // share access
25456|                       FILE_OPEN_IF,
| // create disposition
25457|                       FILE_DIRECTORY_FILE|
25458|
| FILE_OPEN_FOR_BACKUP_INTENT,          // create
| options
25459|                       NULL, // eabuffer
25460|                       0 ); // ealength
25461|
25462|
25463| if(!Status) {
25464|     // set time of directory
25465|     ULONG B = SetFileTime( hFile, (FILETIME*)Time,
| (FILETIME*)Time, (FILETIME*)Time);
25466|     if(B) {
25467|         DLOG((TEXT("Success setting time for
| directory\n")));
25468|     } else {
25469|         DLOG((TEXT("Error %08x setting time for
| directory\n"),GetLastError()));
25470|     }
25471|
25472|     NtClose(hFile);
25473| } else {
25474|     DLOG((TEXT("Error %08x opening
| directory\n"),Status));
25475| }

```

```

25476|    return Status;
25477| }
25478|
25479| #ifdef _DEBUG
25480| DLLEXPORT PSMSTATUS PSMAPI Psm_TestFunction( ULONG
    | Param1, ULONG Param2)
25481| {
25482|     ULONG B=FALSE;
25483|     LONG Err=0;
25484|     OVERLAPPED o;
25485|     ULONG returned;
25486|     pThreadStorage ThreadStorage = GetThreadStorage();
25487|     typedef struct sTest {
25488|         ULONG Param1;
25489|         ULONG Param2;
25490|     } tTest,*pTest;
25491|     tTest Test;
25492|     Test.Param1 = Param1;
25493|     Test.Param2 = Param2;
25494|
25495|     memset(&o,0,sizeof(o));
25496|     o.hEvent = CreateEvent( NULL, FALSE, FALSE, NULL );
25497|
25498|     B = DeviceIoControl(ThreadStorage->PSManHandle,
25499|         IOCTL_TEST_FUNCTION,
25500|         &Test,
25501|         sizeof(Test),
25502|         NULL,
25503|         0,
25504|         &returned,
25505|         &o);
25506|
25507|     if(B) {
25508|         Err = 0;
25509|     } else {
25510|         Err = GetLastError();
25511|         DLOG((TEXT("Error %08x Calling test
    | routine\n"),Err));
25512|     }
25513|     CloseHandle(o.hEvent);
25514|     return Err;
25515| }
25516| #endif
25517|
25518| DLLEXPORT PSMSTATUS PSMAPI Psm_SetClusterRegistry(
    | PVOID SnapShot )
25519| {
25520|     return UpdateClusterRegistries(SnapShot);
25521| }
25522|

```

```

25523|
25524| PSMSTATUS PSMAPI PSMI_LogEvent( ULONG EventId, ULONG
    | Status, ULONG NumStrings, WCHAR *Strings[] )
25525| {
25526|     ULONG B=FALSE;
25527|     LONG Err;
25528|     OVERLAPPED o;
25529|     ULONG returned;
25530|     pThreadStorage ThreadStorage = GetThreadStorage();
25531|     pPSM_LogEvent Log;
25532|     ULONG i;
25533|     ULONG Len;
25534|
25535|     Len = 0;
25536|
25537|     if(NumStrings) {
25538|         for(i=0;i<NumStrings;i++) {
25539|             Len += wcslen(Strings[i]);
25540|         }
25541|
25542|         Len*=sizeof(WCHAR);
25543|         Len+=NumStrings*sizeof(WCHAR); // nulls at end
    | of string
25544|         Len+=NumStrings*sizeof(PVOID); // for pointers
    | to strings
25545|     }
25546|     Len+=sizeof(tPSM_LogEvent);
25547|
25548|     Log = LocalAlloc(LPTR,Len);
25549|     if(Log) {
25550|         PWCHAR p=(PWCHAR)Log->Strings;
25551|         Log->EventId = EventId;
25552|         Log->Status = Status;
25553|         Log->NumStrings = NumStrings;
25554|
25555|         if(NumStrings) {
25556|             p+=NumStrings*sizeof(PVOID);
25557|             for(i=0;i<NumStrings;i++) {
25558|                 wcscpy(p,Strings[i]);
25559|                 Log->Strings[i] = p;
25560|                 p+=wcslen(Strings[i]);
25561|                 *p=0;
25562|                 p++;
25563|             }
25564|         }
25565|
25566|         memset(&o,0,sizeof(o));
25567|         o.hEvent = CreateEvent( NULL, FALSE, FALSE,
    | NULL );
25568|

```

```

25569|     B = DeviceIoControl(
| ThreadStorage->PSManHandle,
25570|     IOCTL_LOG_EVENT,
25571|     Log,
25572|     Len,
25573|     NULL,
25574|     0,
25575|     &returned,
25576|     &o);
25577|
25578|     if(B) {
25579|         Err = 0;
25580|     } else {
25581|         Err = GetLastError();
25582|         DLOG((TEXT("Error %08x Calling
| LogEvent\n"),Err));
25583|     }
25584|     CloseHandle(o.hEvent);
25585| } else {
25586|     Err = ERROR_OUTOFMEMORY;
25587| }
25588| return Err;
25589| }
25590|
25591|
25592| /* adapted from "Data structures and algorithms
| analysis in C"
25593| by Mark Allen Weiss, Second Edition, Addison Wesley,
25594| ISBN:0-201-49840-5 page 397
25595|
25596| original algorithm returned double instead of ULONG
25597| */
25598| STATIC ULONG Random( void )
25599| {
25600|     ULONG TmpSeed;
25601|     int Decimal,Sign;
25602|
25603| // do not change these numbers, see page 394=395 for
| explanation
25604| // but basically these provide a full generation of
| random numbers
25605| // before recycling
25606|
25607| #define A 48271L
25608| #define M 2147483647L
25609| #define Q ( M / A )
25610| #define R ( M % A )
25611|
25612|     TmpSeed = A * (RandomNumberSeed % Q) - R *
| (RandomNumberSeed / Q);

```

```

25613|
25614|     if(TmpSeed>=0)
25615|         RandomNumberSeed = TmpSeed;
25616|     else
25617|         RandomNumberSeed = TmpSeed+M;
25618|
25619|     return atol(_fcvt((1.0*RandomNumberSeed) /
    | M,12,&Decimal,&Sign));
25620| }
25621|
25622| STATIC ULONG CheckKey( ULONG SerialNumber, ULONG
    | CheckCode )
25623| {
25624|     ULONG LoopCount;
25625|     ULONG CCode;
25626|     ULONG i;
25627|
25628|     RandomNumberSeed =
    | (SerialNumber+2)*(SerialNumber+1);
25629|     LoopCount = (SerialNumber+1) % 63;
25630|     for(i=0;i<LoopCount;i++) {
25631|         CCode = Random();
25632|     }
25633|     return (CCode == CheckCode);
25634| }
25635|
25636| /*
25637| ProductType can be one of the following:
25638|
25639| WinNT    = Windows NT workstation
25640| LanmanNT = Windows NT Server domain controller (primary
    | or backup)
25641| ServerNT = Windows NT Server stand-alone
25642|
25643| On Special Editions of Windows NT the ProductSuite key
    | exists and can be the following values:
25644|
25645| Terminal Server = Windows NT Server 4.0 Terminal Server
    | Edition
25646| Enterprise     = Windows NT Server 4.0 Enterprise
    | Edition
25647| */
25648|
25649| STATIC int GetNTSystemType( )
25650| {
25651|     HKEY Key;
25652|     LONG Err=0;
25653|     TCHAR ProductType[80];
25654|     TCHAR ProductSuite[80];
25655|     LONG DataSize=sizeof(ProductType);

```



```

25656|  TCHAR KeyName[256];
25657|  int Type;
25658|
25659|  | _tcsncpy(KeyName,TEXT("SYSTEM\\CurrentControlSet\\Control
    | \\ProductOptions"));
25660|
25661|  // open the registry
25662|  Err = RegOpenKeyEx(
25663|      HKEY_LOCAL_MACHINE, // handle of open key
25664|      KeyName, // address of name of subkey to
    | open
25665|      0, // reserved
25666|      KEY_READ, // security access mask
25667|      &Key// address of handle of open key
25668|  );
25669|  if(Err==0) {
25670|      Err = RegQueryValueEx(
25671|          Key, // handle of key to query
25672|          TEXT("ProductType"), // address of name
    | of value to query
25673|          NULL, // reserved
25674|          NULL, // address of buffer for value type
25675|          (char*)ProductType, // address of data
    | buffer
25676|          &DataSize // address of data buffer size
25677|      );
25678|
25679|      if(Err) {
25680|          DLOG((TEXT("Error %08x getting product
    | type\n"),Err));
25681|      } else {
25682|          DLOG((TEXT("ProductType =
    | '%s\n"),ProductType));
25683|      }
25684|
25685|      DataSize=sizeof(ProductSuite);
25686|
25687|      Err = RegQueryValueEx(
25688|          Key, // handle of key to query
25689|          TEXT("ProductSuite"), // address of name
    | of value to query
25690|          NULL, // reserved
25691|          NULL, // address of buffer for value type
25692|          (char*)ProductSuite, // address of data
    | buffer
25693|          &DataSize // address of data buffer size
25694|      );
25695|
25696|      // close the registry

```

```

25697|     RegCloseKey(Key);
25698| }
25699|
25700| if(Err) {
25701|     DLOG((TEXT("Error %08x getting product
    | suite\n"),Err));
25702|     _tcscpy(ProductSuite,TEXT(""));
25703| } else {
25704|     DLOG((TEXT("ProductSuite =
    | '%s'\n"),ProductSuite));
25705| }
25706|
25707| Type = 0;
25708|
25709| if(_tcsicmp(TEXT("winnt"),ProductType)==0)
25710|     Type |= WINDOWS_NT_WORKSTATION;
25711| else
25712| if(_tcsicmp(TEXT("servernt"),ProductType)==0)
25713|     Type |= WINDOWS_NT_SERVER;
25714| else
25715| if(_tcsicmp(TEXT("lanmannt"),ProductType)==0)
25716|     Type |= WINDOWS_NT_SERVER | WINDOWS_NT_DOMAIN;
25717|
25718| // now check product suite
25719| if(_tcsicmp(TEXT("Terminal
    | Server"),ProductSuite)==0)
25720|     Type |= WINDOWS_NT_TERMINAL_SERVER;
25721| else
25722| if(_tcsicmp(TEXT("Enterprise"),ProductSuite)==0)
25723|     Type |= WINDOWS_NT_ENTERPRISE;
25724|
25725| return Type;
25726|
25727| }
25728|
25729| #define SE_INTERACTIVE_LOGON_NAME
    | TEXT("SeInteractiveLogonRight")
25730| #define SE_NETWORK_LOGON_NAME
    | TEXT("SeNetworkLogonRight")
25731| #define SE_BATCH_LOGON_NAME
    | TEXT("SeBatchLogonRight")
25732| #define SE_SERVICE_LOGON_NAME
    | TEXT("SeServiceLogonRight")
25733|
25734|
25735| STATIC BOOL Login_EnablePrivilege ( LPTSTR Privilege )
25736| {
25737|     BOOL Success;
25738|     HANDLE Token;
25739|     LUID Luid;

```

```

25740|  TOKEN_PRIVILEGES TokenPrivileges;
25741|
25742|  Success =
    | OpenProcessToken(GetCurrentProcess(),TOKEN_ADJUST_PRIVIL
    | EGES | TOKEN_QUERY, &Token );
25743|  if(Success) {
25744|      Success = LookupPrivilegeValue(0, Privilege,
    | &Luid );
25745|      if(Success) {
25746|          TokenPrivileges.PrivilegeCount    = 1;
25747|          TokenPrivileges.Privileges[0].Luid = Luid;
25748|          TokenPrivileges.Privileges[0].Attributes =
    | SE_PRIVILEGE_ENABLED;
25749|          // always returns true
25750|          Success = AdjustTokenPrivileges( Token,
    | FALSE, &TokenPrivileges, 0, 0, 0 );
25751|          if(GetLastError()==ERROR_SUCCESS) {
25752|              Success = TRUE;
25753|          } else {
25754|              DLOG((TEXT("Error %08x setting
    | privileges\n"),GetLastError()));
25755|              Success = FALSE;
25756|          }
25757|      } else {
25758|          DLOG((TEXT("Error %08x looking up privilege
    | value\n"),GetLastError()));
25759|      }
25760|  } else {
25761|      DLOG((TEXT("Error %08x opening process
    | token\n"),GetLastError()));
25762|  }
25763|
25764|  return Success;
25765| }
25766|
25767| STATIC BOOL Login_DisablePrivilege ( LPTSTR Privilege )
25768| {
25769|  BOOL Success;
25770|  HANDLE Token;
25771|  LUID Luid;
25772|  TOKEN_PRIVILEGES TokenPrivileges;
25773|
25774|  Success =
    | OpenProcessToken(GetCurrentProcess(),TOKEN_ADJUST_PRIVIL
    | EGES | TOKEN_QUERY, &Token );
25775|  if(Success) {
25776|      Success = LookupPrivilegeValue(0, Privilege,
    | &Luid );
25777|      if(Success) {
25778|          TokenPrivileges.PrivilegeCount    = 1;

```

```

25779|         TokenPrivileges.Privileges[0].Luid = Luid;
25780|         TokenPrivileges.Privileges[0].Attributes =
| 0;
25781|         // always returns true
25782|         Success = AdjustTokenPrivileges( Token,
| FALSE, &TokenPrivileges, 0, 0, 0 );
25783|         if(GetLastError()==ERROR_SUCCESS) {
25784|             Success = TRUE;
25785|         } else {
25786|             DLOG((TEXT("Error %08x setting
| privileges\n"),GetLastError()));
25787|             Success = FALSE;
25788|         }
25789|     } else {
25790|         DLOG((TEXT("Error %08x looking up privilege
| value\n"),GetLastError()));
25791|     }
25792| } else {
25793|     DLOG((TEXT("Error %08x opening process
| token\n"),GetLastError()));
25794| }
25795|
25796| return Success;
25797| }
25798|
25799| STATIC pSnapShot AllocSnapShot()
25800| {
25801|     pSnapShot SnapShot=NULL;
25802|     ULONG i;
25803|
25804|     WaitForSingleObject ( SharedMemoryMutex, INFINITE
| );
25805|     __try {
25806|         for(i=0;i<MAX_NUMBER_OF_SNAPSHOTS;i++) {
25807|             if(!(SharedMemory->SnapShotsInUse[i/8] & (1
| << (i % 8)))) {
25808|                 CHAR Save =
| SharedMemory->SnapShotsInUse[i/8];
25809|                 // Set bit
25810|                 SharedMemory->SnapShotsInUse[i/8] |= (1
| << (i % 8));
25811|                 SnapShot =
| &SharedMemory->ActualSnapShotSpace[i];
25812|                 DLOG(("%08x: Set bit %d (%d %d) (%02x
| %02x)\n",SnapShot,i,i/8,i%8,Save,SharedMemory->SnapShots
| InUse[i/8]));
25813|                 break;
25814|             }
25815|         }
25816|     } __finally {

```

```

25817|     ReleaseMutex(SharedMemoryMutex);
25818| }
25819| if(!SnapShot) {
25820|     DLOG(("Out of memory in global namespace!\n"));
25821|     SetLastError(ERROR_OUTOFMEMORY);
25822| }
25823| return SnapShot;
25824| }
25825|
25826| STATIC void FreeSnapShot( pSnapShot SnapShot )
25827| {
25828|     ULONG i;
25829|     if(SnapShot) {
25830|         WaitForSingleObject ( SharedMemoryMutex,
25831|             | INFINITE );
25832|         __try {
25833|             for(i=0;i<MAX_NUMBER_OF_SNAPSHOTS;i++) {
25834|                 if(SnapShot ==
25835|                     | &SharedMemory->ActualSnapShotSpace[i]) {
25836|                     CHAR Save =
25837|                         | SharedMemory->SnapShotsInUse[i/8];
25838|                     // clear bit
25839|                     SharedMemory->SnapShotsInUse[i/8]
25840|                         | &= ~(1 << (i % 8));
25841|                     DLOG(("08x: Clr bit %d (%d %d)
25842|                         | (%02x
25843|                         | %02x)\n",SnapShot,i,i/8,i%8,Save,SharedMemory->SnapShots
25844|                         | InUse[i/8]));
25845|                     | memset(SnapShot,0,sizeof(tSnapShot));
25846|                     SnapShot=NULL;
25847|                     break;
25848|                 }
25849|             }
25850|             if(SnapShot) {
25851|                 DLOG(("Pointer 08x not
25852|                     | found!\n",SnapShot));
25853|             }
25854|         } __finally {
25855|             ReleaseMutex(SharedMemoryMutex);
25856|         }
25857|     }
25858| }
25859|
25860| STATIC ULONG IsValidSnapShot( pSnapShot SnapShot )
25861| {
25862|     ULONG i;
25863|     ULONG B=FALSE;
25864|     if(SnapShot) {

```

```

25858|     WaitForSingleObject ( SharedMemoryMutex,
| INFINITE );
25859|     __try {
25860|         for(i=0;i<MAX_NUMBER_OF_SNAPSHOTS;i++) {
25861|             if(SnapShot ==
| &SharedMemory->ActualSnapShotSpace[i]) {
25862|
| if(SharedMemory->SnapShotsInUse[i/8] & (1 << (i % 8)))
| {
25863|                 if(SnapShot->OwningThread ==
| GetThreadStorage()) {
25864|                     B = TRUE;
25865|                     break;
25866|                 }
25867|             }
25868|         }
25869|     }
25870| } __finally {
25871|     ReleaseMutex(SharedMemoryMutex);
25872| }
25873| }
25874| return B;
25875| }
25876|
25877| //-----
| -----
25878|
25879|
25880| DLLEXPORT PSMSTATUS PSMAPI Psm_GetActiveSnapshotTable (
25881| OUT ULONG      *numberOfActiveSnapshots,
25882| OUT pSnapShot  snapshotTable[],
25883| IN ULONG      snapshotTableSlots /*max number of
| pointers 'snapshotTable' can hold*/ )
25884| {
25885|     PSMSTATUS err = 0;
25886|
25887|     if ( numberOfActiveSnapshots && snapshotTable &&
| snapshotTableSlots>0 ) {
25888|         WaitForSingleObject ( SharedMemoryMutex,
| INFINITE );
25889|         __try {
25890|             ULONG counter=0, i=0, j=0;
25891|
25892|             // First pass... enumerate temporary
| snapshots:
25893|             for ( i=0; i < MAX_NUMBER_OF_SNAPSHOTS; ++i
| ) {
25894|                 if ( (SharedMemory->SnapShotsInUse[i/8]
| & (1 << (i % 8))) ) {
25895|                     pSnapShot snapshot =

```

```

    | &SharedMemory->ActualSnapShotSpace[i];
25896|         PVOID kernelPointer =
    | snapshot->KernelSnapShotPointer;
25897|         ULONG isUnique = TRUE;
25898|
25899|         for ( j=0; j<counter && isUnique;
    | ++j ) {
25900|             if ( kernelPointer ==
    | snapshotTable[j]->KernelSnapShotPointer ) {
25901|                 isUnique = FALSE;
25902|             }
25903|         }
25904|
25905|         if ( isUnique ) {
25906|             if ( counter <
    | snapshotTableSlots-1 ) {
25907|                 snapshotTable[counter++] =
    | snapshot;
25908|             } else {
25909|                 err = ERROR_MORE_DATA;
25910|                 break;
25911|             }
25912|         }
25913|     }
25914| }
25915|
25916| if ( !err ) {
25917|     // Second pass... append all persistent
    | snapshots:
25918|     if ( counter <= snapshotTableSlots ) {
25919|         err = Psm_GetPersistentSnapShots (
25920|             &snapshotTable[counter],
25921|             sizeof(pSnapShot) *
    | (snapshotTableSlots - counter) );
25922|
25923|         if ( !err ) {
25924|             // figure out how many are in
    | the array by looking for NULL termination.
25925|             while ( snapshotTable[counter]
    | != NULL ) {
25926|                 if ( counter ==
    | snapshotTableSlots-1 ) {
25927|                     err = ERROR_MORE_DATA;
25928|                     break;
25929|                 }
25930|
25931|                 ++counter;
25932|             }
25933|         }
25934|     } else {

```

```

25935|          // sanity check: This should never
| happen if first pass code is correct.
25936|          err = ERROR_MORE_DATA;
25937|          DLOG(("!!! Sanity check failed in
| Psm_GetActiveSnapshotTable: s=%lu
| c=%lu\n", snapshotTableSlots, counter));
25938|      }
25939|  }
25940|
25941|      *numberOfActiveSnapshots = counter;
25942|  } __finally {
25943|      ReleaseMutex(SharedMemoryMutex);
25944|  }
25945|  } else {
25946|      err = ERROR_INVALID_PARAMETER;
25947|  }
25948|
25949|  return err;
25950| }
25951|
25952| //-----
| -----
25953|
25954|
25955| STATIC WCHAR *FNVFS_ListOfVolumeNames = NULL;
25956| STATIC ULONG FNVFS_ListOfVolumeNamesSizeInBytes = 0;
25957|
25958|
25959| DLLEXPORT PSMSTATUS PSMAPI
| Psm_FindNextVolumeForSnapshot (
25960|  IN pSnapShot  snapshot,
25961|  OUT WCHAR     **originalVolumeName,
25962|  IN OUT ULONG  *iteratorState )
25963| {
25964|  PSMSTATUS err=0;
25965|  ULONG i=0;
25966|
25967|  if ( snapshot && originalVolumeName &&
| iteratorState ) {
25968|      *originalVolumeName = NULL;
25969|      WaitForSingleObject ( SharedMemoryMutex,
| INFINITE );
25970|      __try {
25971|          if (
| Psm_IsPersistentSnapShotPointer(snapshot) ) {
25972|              // It's a fancy new persistent
| snapshot...
25973|              if ( *iteratorState == 0 ) {
25974|                  // This is the FindFirst call, so
| get a list of all volumes up front.

```



```

25975|          // For each subsequent call, we
| keep re-using the list.
25976|          if ( FNVFS_ListOfVolumeNames ==
| NULL ) {
25977|          | FNVFS_ListOfVolumeNamesSizeInBytes = 32 * 1024 *
| sizeof(WCHAR);
25978|          FNVFS_ListOfVolumeNames =
| LocalAlloc (LPTR, FNVFS_ListOfVolumeNamesSizeInBytes);
25979|          if ( FNVFS_ListOfVolumeNames ==
| NULL ) {
25980|              err = ERROR_OUTOFMEMORY;
25981|
| FNVFS_ListOfVolumeNamesSizeInBytes = 0;
25982|          }
25983|      }
25984|
25985|      if ( FNVFS_ListOfVolumeNames !=
| NULL ) {
25986|          err =
| Psm_GetKernelSnapShotVolumesW (
25987|              snapshot,
25988|              FNVFS_ListOfVolumeNames,
25989|
| FNVFS_ListOfVolumeNamesSizeInBytes );
25990|      }
25991|  }
25992|
25993|      if ( !err ) {
25994|          if ( FNVFS_ListOfVolumeNames ==
| NULL ) {
25995|              err = ERROR_OUTOFMEMORY;
25996|
| FNVFS_ListOfVolumeNamesSizeInBytes = 0;
25997|          } else {
25998|              WCHAR *nthString =
| FNVFS_ListOfVolumeNames;
25999|              for ( i=0; i < *iteratorState
| && *nthString; ++i ) {
26000|                  nthString +=
| wcslen(nthString) + 1;
26001|              }
26002|
26003|              if ( *nthString ) {
26004|                  *originalVolumeName =
| nthString;
26005|                  ++(*iteratorState);
26006|              } else {
26007|                  *originalVolumeName = NULL;
26008|              }

```

```

26009|         }
26010|     }
26011| } else {
26012|     // It's an old-style temporary
    | snapshot...
26013|     for ( i = *iteratorState; i <
    | SharedMemory->NumberOfMappedVolumes; ++i ) {
26014|         tMapDrive *volume =
    | &SharedMemory->MappedVolumes[i];
26015|         pSnapShot volSnapshot =
    | MakePointer(volume->SnapShotOffset);
26016|         if ( volSnapshot == snapshot ) {
26017|             *originalVolumeName =
    | volume->OriginalUserName;
26018|             *iteratorState = i + 1;
26019|             break;
26020|         }
26021|     }
26022| }
26023| } __finally {
26024|     ReleaseMutex(SharedMemoryMutex);
26025| }
26026| } else {
26027|     err = ERROR_INVALID_PARAMETER;
26028| }
26029|
26030| return err;
26031| }
26032|
26033|
26034| //-----
    | -----
26035|
26036|
26037| DLLEXPORT PSMSTATUS PSMAPI
    | Psm_FindFirstVolumeForSnapshot (
26038|     IN pSnapShot  snapshot,
26039|     OUT WCHAR     **originalVolumeName,
26040|     OUT ULONG     *iteratorState )
26041| {
26042|     PSMSTATUS err = 0;
26043|
26044|     if ( snapshot && originalVolumeName &&
    | iteratorState ) {
26045|         *iteratorState = 0;
26046|         err = Psm_FindNextVolumeForSnapshot ( snapshot,
    | originalVolumeName, iteratorState );
26047|     } else {
26048|         err = ERROR_INVALID_PARAMETER;
26049|     }

```

```

26050|
26051|     return err;
26052| }
26053|
26054|
26055|
26056| //-----
| -----
26057|
26058|
26059| DWORD ExceptionFilter( EXCEPTION_POINTERS *ep )
26060| {
26061|     ULONG ret;
26062|
26063|     DLOG(("Exception occured\n"));
26064|     DLOG(("Exception:
| %08x\n", ep->ExceptionRecord->ExceptionCode));
26065|     DLOG(("Flags   :
| %08x\n", ep->ExceptionRecord->ExceptionFlags));
26066|     DLOG(("Address :
| %08x\n", ep->ExceptionRecord->ExceptionAddress));
26067|
26068|     switch ( ep->ExceptionRecord->ExceptionCode ) {
26069|
26070|         case EXCEPTION_ACCESS_VIOLATION :
26071|             DLOG(("Attempted to %s invalid memory at
| %08x\n",
26072|
| ep->ExceptionRecord->ExceptionInformation[0] ?
| TEXT("write") : TEXT("read"),
26073|
| ep->ExceptionRecord->ExceptionInformation[1]
26074|             ));
26075|
26076|         case EXCEPTION_ARRAY_BOUNDS_EXCEEDED :
26077|             ret = EXCEPTION_EXECUTE_HANDLER;
26078|             break;
26079|
26080|         case EXCEPTION_BREAKPOINT :
26081|         case EXCEPTION_DATATYPE_MISALIGNMENT :
26082|         case EXCEPTION_FLT_DENORMAL_OPERAND :
26083|         case EXCEPTION_FLT_DIVIDE_BY_ZERO :
26084|         case EXCEPTION_FLT_INEXACT_RESULT :
26085|         case EXCEPTION_FLT_INVALID_OPERATION :
26086|         case EXCEPTION_FLT_OVERFLOW :
26087|         case EXCEPTION_FLT_STACK_CHECK :
26088|         case EXCEPTION_FLT_UNDERFLOW :
26089|         case EXCEPTION_ILLEGAL_INSTRUCTION :
26090|         case EXCEPTION_IN_PAGE_ERROR :
26091|         case EXCEPTION_INT_DIVIDE_BY_ZERO :

```

```

26092|     case EXCEPTION_INT_OVERFLOW :
26093|     case EXCEPTION_INVALID_DISPOSITION :
26094|     case EXCEPTION_NONCONTINUABLE_EXCEPTION :
26095|     case EXCEPTION_PRIV_INSTRUCTION :
26096|     case EXCEPTION_SINGLE_STEP :
26097|     case EXCEPTION_STACK_OVERFLOW :
26098| default:
26099|     // pass to debugger or system
26100|     ret = EXCEPTION_CONTINUE_SEARCH;
26101|     break;
26102|     // EXCEPTION_CONTINUE_SEARCH = continue looking
    | for a error handler
26103|     // EXCEPTION_EXECUTE_HANDLER = execute
    | exception handler
26104|     // EXCEPTION_CONTINUE_EXECUTION = continue as
    | though nothing happened
26105|
26106| }
26107| #ifdef _DEBUG
26108|     if(DebugMode) {
26109|         DebugBreak();
26110|     }
26111| #endif
26112|     return ret;
26113| }
26114|
26115| STATIC int ExecuteFile ( WCHAR *FileName, WCHAR
    | *CommandLine, WCHAR *ChildUser, WCHAR *ChildPassword )
26116| {
26117|     STARTUPINFO StartupInfo={0};
26118|     PROCESS_INFORMATION ProcessInformation={0};
26119|     WCHAR CurrentDir[256]={0};
26120|     WCHAR Comm[256]={0};
26121|     BOOL UserSpecified=FALSE;
26122|     BYTE sidBuffer[100]={0};
26123|     PSID psid=(PSID)&sidBuffer;
26124|     ULONG sidLen=sizeof(sidBuffer);
26125|     WCHAR DomainName[128]={0};
26126|     ULONG DomainNameLen = 128;
26127|     SID_NAME_USE snu={0};
26128|     BOOL B;
26129|     ULONG Err=0;
26130|     LPWSTR pdc=NULL;
26131|     HANDLE Token;
26132|     WCHAR Ext[4]={0};
26133|     HANDLE DirHandle;
26134|     WIN32_FIND_DATA FindFileData;
26135|     WCHAR *p;
26136|     WCHAR FullFile[256];
26137|     pThreadStorage ThreadStorage = GetThreadStorage();

```

```

26138|
26139| GetCurrentDirectoryW( 256, CurrentDir );
26140|
26141| p = wcsrchr(FileName,TEXT("\\"));
26142|
26143| // if a directory is specified
26144| if((p) || (FileName[1]==L':')) {
26145|     // save the directory
26146|     if(!p) p=FileName+2;
26147|     wcsncpy(CurrentDir,FileName,p-FileName);
26148|     CurrentDir[p-FileName] = 0;
26149|
26150|     p = wcschr(FileName,L'.');
26151|     wcsncpy(FullFile,FileName);
26152|     if(p) {
26153|         // if a extention specified. no need to
        | look for it
26154|         DirHandle = FindFirstFileW( FullFile,
        | &FindFileData);
26155|         if(DirHandle!=INVALID_HANDLE_VALUE) {
26156|             FindClose(DirHandle);
26157|             goto RunFile;
26158|         }
26159|     }
26160| } else {
26161|     // no directory specified, use current
26162|
        | swprintf(FullFile,L"%s\\%s",CurrentDir,FileName);
26163| }
26164|
26165| // see if any pifs to run
26166| swprintf(Comm,L"%s.pif",FullFile);
26167| DirHandle = FindFirstFileW( Comm, &FindFileData);
26168| if(DirHandle!=INVALID_HANDLE_VALUE) {
26169|     FindClose(DirHandle);
26170|     goto RunFile;
26171| }
26172|
26173| // check for os/2 files
26174| swprintf(Comm,L"%s.cmd",FullFile);
26175| DirHandle = FindFirstFileW( Comm, &FindFileData);
26176| if(DirHandle!=INVALID_HANDLE_VALUE) {
26177|     FindClose(DirHandle);
26178|     goto RunFile;
26179| }
26180|
26181| // check for batch files.
26182| swprintf(Comm,L"%s.bat",FullFile);
26183| DirHandle = FindFirstFileW( Comm, &FindFileData);
26184| if(DirHandle!=INVALID_HANDLE_VALUE) {

```

```

26185|     FindClose(DirHandle);
26186|     goto RunFile;
26187| }
26188|
26189| // search for exe's
26190| swprintf(Comm,L"%s.exe",FullFile);
26191| DirHandle = FindFirstFileW( Comm, &FindFileData);
26192| if(DirHandle!=INVALID_HANDLE_VALUE) {
26193|     FindClose(DirHandle);
26194|     goto RunFile;
26195| }
26196|
26197| // search for .com files
26198| swprintf(Comm,L"%s.com",FullFile);
26199| DirHandle = FindFirstFileW( Comm, &FindFileData);
26200| if(DirHandle!=INVALID_HANDLE_VALUE) {
26201|     FindClose(DirHandle);
26202|     goto RunFile;
26203| }
26204|
26205| // see if any file to run.
26206| swprintf(Comm,L"%s.*",FullFile);
26207| DirHandle = FindFirstFileW( Comm, &FindFileData);
26208| if(DirHandle==INVALID_HANDLE_VALUE) {
26209| //     DLOG((TEXT("File not found\n")));
26210|     swprintf(Comm,L"%s %s",FileName,CommandLine);
26211|     goto RunFile2;
26212| }
26213| FindClose(DirHandle);
26214|
26215| RunFile:
26216| p = wcsrchr(FindFileData.cFileName,L'.');
26217| if(p) {
26218|     if(_wcsnicmp(p+1,L"bat",3)==0) {
26219|         // a batch file, so run cmd.exe first.
26220|         swprintf(Comm,L"cmd.exe /c \"%s\" %s",
26221| | FindFileData.cFileName,CommandLine);
26222|     } else {
26223|         // not a batch file
26224|         swprintf(Comm,L "\"%s\"
26225| | %s",FindFileData.cFileName,CommandLine);
26226|     }
26227| } else {
26228|     // no extension found
26229|     swprintf(Comm,L "\"%s\"
26230| | %s",FindFileData.cFileName,CommandLine);
26231| }
26232|
26233| RunFile2:
26234| DLOG((TEXT("Going to run file '%S'\n"),Comm));

```

```

26232|
26233|
26234|     if (wcscmp(ChildUser,L"")!=0) {
26235|         UserSpecified = TRUE;
26236|     } else {
26237|         UserSpecified = FALSE;
26238|     }
26239|
26240|     memset(&StartupInfo,0,sizeof(StartupInfo));
26241|     StartupInfo.cb = sizeof(StartupInfo);
26242|     StartupInfo.dwFlags = STARTF_USESHOWWINDOW;
26243|     StartupInfo.wShowWindow = SW_HIDE |
        | SW_SHOWMINIMIZED | SW_MINIMIZE | SW_FORCEMINIMIZE;
26244|     | memset(&ProcessInformation,0,sizeof(ProcessInformation))
        | ;
26245|
26246|     if(UserSpecified) {
26247|         // get the domain of the user...
26248|         B = LookupAccountNameW( NULL,
26249|                                ChildUser,
26250|                                psid,
26251|                                &sidLen,
26252|                                DomainName,
26253|                                &DomainNameLen,
26254|                                &snu );
26255|
26256|         DLOG((TEXT("Found = %s, Name=%S,
        | Domain=%S\n"),B ? TEXT("True") :
        | TEXT("False"),ChildUser,DomainName));
26257|
26258|         // see if valid password it is case sensitive
26259|         if(B) {
26260|             WCHAR *Domain    = DomainName;
26261|             ULONG LogonType   =
        | LOGON32_LOGON_INTERACTIVE;
26262|             ULONG LogonProvider =
        | LOGON32_PROVIDER_DEFAULT;
26263|
26264|             Token = 0;
26265|             Err = 0;
26266|
26267|             // make sure this privilege is on...
26268|             | Login_EnablePrivilege(SE_INTERACTIVE_LOGON_NAME);
26269|             | Login_EnablePrivilege(SE_NETWORK_LOGON_NAME);
26270|             Login_EnablePrivilege(SE_BATCH_LOGON_NAME);
26271|             | Login_EnablePrivilege(SE_SERVICE_LOGON_NAME);

```

```

26272|
26273|         // loop through possible logon types,
        | hopefully
26274|         // one of them will work.
26275| DoLogonAgain:
26276|         B = LogonUserW( ChildUser, Domain,
        | ChildPassword, LogonType, LogonProvider, &Token );
26277|         if(!B) {
26278|             Err = GetLastError();
26279|             DLOG((TEXT("Error %08x during log on
        | for user '%S' in domain '%S'\n"),Err, ChildUser,
        | Domain)));
26280|             if(LogonType<6) {
26281|                 LogonType++;
26282|                 goto DoLogonAgain;
26283|             }
26284|             // failed all attempts to log on
26285|             B = TRUE; // so we dont get 2 error
        | messages
26286|             //goto RunChildAsMe;
26287|         } else {
26288|             DLOG((TEXT("Success on LogonUser\n")));
26289|             Err = 0;
26290|
26291|             // make sure this privilege is on...
26292|             | if(!Login_EnablePrivilege(SE_ASSIGNPRIMARYTOKEN_NAME))
26293|                 DLOG((TEXT("Error Enabling
        | Privilege
        | SE_ASSIGNPRIMARYTOKEN_NAME\n"),GetLastError()));
26294|             | if(!Login_EnablePrivilege(SE_INCREASE_QUOTA_NAME))
26295|                 DLOG((TEXT("Error Enabling
        | Privilege SE_INCREASE_QUOTA_NAME\n"),GetLastError()));
26296|             if(!Login_EnablePrivilege(SE_TCB_NAME))
26297|                 DLOG((TEXT("Error Enabling
        | Privilege SE_TCB_NAME\n"),GetLastError()));
26298|
26299|             DLOG((TEXT("Running file as '%S' in
        | domain '%S'\n"),ChildUser,Domain));
26300|             DLOG((TEXT("Command
        | line='%S'\n"),Comm));
26301|             DLOG((TEXT("Current
        | Dir='%S'\n"),CurrentDir));
26302|
26303|             B = CreateProcessAsUserW(
26304|                 Token,
26305|                 NULL, // pointer to name of
        | executable module
26306|                 Comm, // pointer to command line

```



```

    | string
26307|          NULL, // pointer to process
    | security attributes
26308|          NULL, // pointer to thread
    | security attributes
26309|          FALSE, // handle inheritance flag
26310|          NORMAL_PRIORITY_CLASS, // creation
    | flags
26311|          NULL, // pointer to new
    | environment block
26312|          CurrentDir, // pointer to current
    | directory name
26313|          &StartupInfo, // pointer to
    | STARTUPINFO
26314|          &ProcessInformation // pointer
    | to PROCESS_INFORMATION
26315|          );
26316|
26317| WaitForApp:
26318|          if(!B) {
26319|              Err = GetLastError();
26320|          } else {
26321|              HANDLE Handles[2];
26322|
26323|              // wait for app to finish running,
    | or timeout
26324|              Handles[0] =
    | ProcessInformation.hProcess;
26325|              Handles[1] =
    | ThreadStorage->AbortEvent;
26326|              Err = WaitForMultipleObjects(
    | ThreadStorage->AbortEvent==INVALID_HANDLE_VALUE ? 1 :
    | 2, Handles, FALSE, INFINITE);
26327|              if(Err==WAIT_OBJECT_0) {
26328|                  // Process is gone
26329|                  DLOG((TEXT("Process is
    | done\n"))));
26330|                  Err = 0;
26331|              } else
26332|              if(Err==WAIT_OBJECT_0+1) {
26333|                  // Abort event hit
26334|                  DLOG((TEXT("Abort event
    | signalled during wait for process\n"))));
26335|                  Err = 0;
26336|                  B = FALSE;
26337|              } else {
26338|                  DLOG((TEXT("Error %08x in
    | Wait\n"),Err));
26339|                  B = FALSE;
26340|              }

```

```

26341|
26342|          // dont need the handles any more.
26343|
26344|          | CloseHandle(ProcessInformation.hThread);
26345|          | CloseHandle(ProcessInformation.hProcess);
26346|          }
26347|          }
26348|          // log off user.(if logged on)
26349|          if(Token != INVALID_HANDLE_VALUE)
26350|              CloseHandle(Token);
26351|          } else {
26352|              Err = GetLastError();
26353|              DLOG((TEXT("Error %08x looking up
26354|              | user\n"),Err));
26355|              B = TRUE; // so we dont get 2 messages
26356|              // fail execute so user can debug.
26357|              //goto RunChildAsMe;
26358|          } else {
26359| //RunChildAsMe:
26360|          DLOG((TEXT("Running file as me\n")));
26361|          DLOG((TEXT("Command line='%S'\n"),Comm));
26362|          DLOG((TEXT("Current Dir='%S'\n"),CurrentDir));
26363|
26364|          B = CreateProcessW(
26365|              NULL, // pointer to name of executable
26366|              | module
26367|              Comm, // pointer to command line string
26368|              NULL, // pointer to process security
26369|              | attributes
26370|              NULL, // pointer to thread security
26371|              | attributes
26372|              FALSE, // handle inheritance flag
26373|              NORMAL_PRIORITY_CLASS, // creation flags
26374|              NULL, // pointer to new environment block
26375|              CurrentDir, // pointer to current directory
26376|              | name
26377|              &StartupInfo, // pointer to STARTUPINFO
26378|              &ProcessInformation // pointer to
26379|              | PROCESS_INFORMATION
26380|              );
26381|          Token = INVALID_HANDLE_VALUE;
26382|          goto WaitForApp;
26383|          }
26384|
26385|          if(!B) {
26386|              DLOG((TEXT("Error %08x executing file
26387|              | '%S'\n"),Err,Comm));

```

```

26382|    }
26383|    return Err;
26384| }
26385|
26386| STATIC void InitForNewThread()
26387| {
26388|     pThreadStorage ThreadStorage =
        | TlsGetValue(TlsIndex);
26389|
26390|     if(ThreadStorage) {
26391|         memset(ThreadStorage,0,sizeof(tThreadStorage));
26392|
26393|         ThreadStorage->PSManEvent = CreateEvent( NULL,
        | FALSE, FALSE, NULL );
26394|         if
        | (ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) {
26395|             PSMI_OpenManager(
        | &ThreadStorage->PSManHandle );
26396|
        | if(ThreadStorage->PSManHandle==INVALID_HANDLE_VALUE) {
26397|                 DLOG((TEXT("Unable to open psm\n")));
26398|                 CloseHandle(ThreadStorage->PSManEvent);
26399|                 ThreadStorage->PSManEvent =
        | INVALID_HANDLE_VALUE;
26400|             } else {
26401|                 //DLOG((TEXT("Psm open\n")));
26402|                 ThreadStorage->NumOpens    = 0;
26403|                 ThreadStorage->NumSnapShots = 0;
26404|                 ThreadStorage->RegisterCalled = 0;
26405|                 ThreadStorage->UsedTempFile = FALSE;
26406|             }
26407| #ifdef _DEBUG
26408|             {
26409|                 TCHAR Filename[80];
26410|
26411|                 _tmkdir(TEXT("psm"));
26412|
        | _sprintf(Filename,TEXT("psm\\%d"),GetCurrentProcessId()
        | );
26413|                 _tmkdir(Filename);
26414|
        | _sprintf(Filename,TEXT("psm\\%d\\%d.log"),GetCurrentPro
        | cessId(),GetCurrentThreadId());
26415|                 ThreadStorage->DebugLogFile=CreateFile(
        | Filename,
26416|                 GENERIC_WRITE,
26417|
        | FILE_SHARE_READ|FILE_SHARE_WRITE,
26418|                 NULL,
26419|                 CREATE_ALWAYS,

```

```

26420|
    | FILE_FLAG_WRITE_THROUGH,
26421|             NULL);
26422|
26423|     }
26424| #endif
26425|     }
26426| }
26427| }
26428|
26429| /*
26430| When a process uses load-time linking with this DLL,
    | the entry-point
26431| function is sufficient to manage the thread local
    | storage. Problems can
26432| occur with a process that uses run-time linking because
    | the entry-point
26433| function is not called for threads that exist before
    | the LoadLibrary
26434| function is called, so TLS memory is not allocated for
    | these threads.
26435| The following example solves this problem by checking
    | the value returned
26436| by the TlsGetValue function and allocating memory if
    | the value indicates
26437| that the TLS slot for this thread is not set.
26438| */
26439| STATIC pthread_storage GetThreadStorage()
26440| {
26441|     pthread_storage ThreadStorage =
        | TlsGetValue(TlsIndex);
26442|
26443|     // If NULL, allocate memory for this thread.
26444|
26445|     if (ThreadStorage == NULL) {
26446|         ThreadStorage = LocalAlloc(LPTR,
            | sizeof(pthread_storage));
26447|         TlsSetValue(TlsIndex, ThreadStorage);
26448|         InitForNewThread();
26449|     }
26450|     return ThreadStorage;
26451| }
26452|
26453| STATIC ULONG AddDirToNotBackup( WCHAR *Dir, ULONG
    | IncludeSubDirs )
26454| {
26455|     HKEY Key;
26456|     ULONG Err;
26457|     TCHAR RegStr[255];
26458|     WCHAR Line[300];

```

```

26459|
26460|
26461|     | _stprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Contro
26462|     | \\BackupRestore\\FilesNotToBackup"));
26463| // open the registry
26464| Err = RegOpenKeyEx(
26465|     HKEY_LOCAL_MACHINE, // handle of open key
26466|     RegStr, // address of name of subkey to open
26467|     0, // reserved
26468|     KEY_READ | KEY_WRITE, // security access mask
26469|     &Key// address of handle of open key
26470| );
26471| if(!Err) {
26472|     ULONG Len;
26473|     wcscpy(Line,L"\\");
26474|     wcscat(Line,Dir);
26475|     wcscat(Line,L"\\*");
26476|     if(IncludeSubDirs) {
26477|         wcscat(Line,L" /s");
26478|     }
26479|     // double null terminate
26480|     Len = wcslen(Line);
26481|     Line[Len]=0;
26482|     Line[Len+1] = 0;
26483|     Line[Len+2] = 0;
26484|     Err = RegSetValueExW(Key,L"Persistent Storage
26485|     | Manager
26486|     | (Images)",0,REG_MULTI_SZ,(PCHAR)Line,(Len*sizeof(WCHAR))
26487|     | +(sizeof(WCHAR)*2));
26488|     if(Err) {
26489|         DLOG(("Error %08x setting backup
26490|         | key\n",Err));
26491|     }
26492|     RegCloseKey(Key);
26493| } else {
26494|     DLOG(("Error %08x opening backup key\n",Err));
26495| }
26496| return Err;
26497| }
26498|
26499| STATIC void GetRegistrySettings()
26500| {
26501|     ULONG DataSize;
26502|     HKEY Key;
26503|     ULONG Err;
26504|     ULONG Add=0;
26505|     TCHAR RegStr[255];

```

```

    | _sprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
    | es\\PSMan%d"), NtBuildNumber > 1381 ? 5 : 4);
26503| // open the registry
26504| Err = RegOpenKeyEx(
26505|     HKEY_LOCAL_MACHINE, // handle of open key
26506|     RegStr, // address of name of subkey to open
26507|     0, // reserved
26508|     KEY_READ, // security access mask
26509|     &Key// address of handle of open key
26510| );
26511|
26512| if(Err==0) {
26513|     DataSize = 256;
26514|     Err = RegQueryValueExW(
26515|         Key, // handle of key to query
26516|         L"CacheFileLocation", // address of name
    | of value to query
26517|         NULL, // reserved
26518|         NULL, // address of buffer for value type
26519|         (char*)CacheFileLocation, // address of
    | data buffer
26520|         &DataSize // address of data buffer size
26521|     );
26522|     CacheFileLocationKeyExists = (Err == 0);
26523|
26524| #ifdef _DEBUG
26525|     DataSize = 4;
26526|     Err = RegQueryValueEx(
26527|         Key, // handle of key to query
26528|         TEXT("DebugLevel"), // address of name of
    | value to query
26529|         NULL, // reserved
26530|         NULL, // address of buffer for value type
26531|         (char*)&DebugMode, // address of data
    | buffer
26532|         &DataSize // address of data buffer size
26533|     );
26534| #endif
26535| // close the registry
26536| RegCloseKey(Key);
26537| }
26538|
26539| // read persistent registry entries
26540|
26541|
    | _sprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
    | es\\PSMan%d\\persistent"), NtBuildNumber > 1381 ? 5 :
    | 4);
26542| // open the registry
26543| Err = RegOpenKeyEx(

```

```

26544|     HKEY_LOCAL_MACHINE, // handle of open key
26545|     RegStr, // address of name of subkey to open
26546|     0, // reserved
26547|     KEY_READ, // security access mask
26548|     &Key// address of handle of open key
26549| );
26550|
26551| if(Err==0) {
26552|     DataSize = 256;
26553|     Err = RegQueryValueExW(
26554|         Key, // handle of key to query
26555|         L"SnapShotLocation", // address of name
        | of value to query
26556|         NULL, // reserved
26557|         NULL, // address of buffer for value type
26558|         (char*)SnapShotLocation, // address of
        | data buffer
26559|         &DataSize // address of data buffer size
26560|     );
26561|     if(Err) {
26562|         wcscpy(SnapShotLocation,L"snapshots");
26563|     }
26564|     DataSize = 4;
26565|     Err = RegQueryValueExW(
26566|         Key, // handle of key to query
26567|         L"AddDoNotBackupDir", // address of name
        | of value to query
26568|         NULL, // reserved
26569|         NULL, // address of buffer for value type
26570|         (char*)Add, // address of data buffer
26571|         &DataSize // address of data buffer size
26572|     );
26573|
26574|     // if not there, then dont add
26575|     if(!Err) {
26576|         if(Add) {
26577|             // add this directory to the list of
        | files/dirs to not backup
26578|
        | AddDirToNotBackup(SnapShotLocation,TRUE);
26579|         }
26580|     }
26581|
26582|     DataSize=255;
26583|     Err = RegQueryValueExW(
26584|         Key, // handle of key to query
26585|         L"SnapShotPattern", // address of name of
        | value to query
26586|         NULL, // reserved
26587|         NULL, // address of buffer for value type

```

```

26588|         (char*)SnapShotPattern, // address of data
      | buffer
26589|         &DataSize // address of data buffer size
26590|     );
26591|     if(Err) {
26592|         wcscpy(SnapShotPattern,L"snapshot.%i");
26593|     }
26594|
26595|     // close the registry
26596|     RegCloseKey(Key);
26597| } else {
26598|     // set defaults if persistent key doesnt exist
26599|     wcscpy(SnapShotLocation,L"snapshots");
26600|     wcscpy(SnapShotPattern,L"snapshot.%i");
26601| }
26602|
26603| }
26604|
26605| STATIC ULONG CheckForZeroTerminatorA( char *Str, ULONG
      | Len )
26606| {
26607|     ULONG i=0;
26608|
26609|     while((i<Len) && (Str[i]!='\0')) {
26610|         i++;
26611|     }
26612|     return Str[i]=='\0' ? TRUE : FALSE;
26613| }
26614|
26615| STATIC ULONG CheckForZeroTerminatorW( WCHAR *Str, ULONG
      | Len )
26616| {
26617|     ULONG i=0;
26618|
26619|     while((i<Len) && (Str[i]!=L'\0')) {
26620|         i++;
26621|     }
26622|     return Str[i]==L'\0' ? TRUE : FALSE;
26623| }
26624|
26625| // support for installshield install scripts
26626| __declspec( dllexport ) LONG __cdecl
      | UninstInitialize(HWND hwndDlg,HANDLE hInstance,LONG
      | IRes)
26627| {
26628|     BOOLEAN NeedsRebooting=FALSE;
26629|     return Psm_UnInstallPsm(&NeedsRebooting);
26630| }
26631|
26632| __declspec( dllexport ) LONG __cdecl

```



```

    | UninstUnInitialize(HWND hwndDlg,HANDLE hInstance,LONG
    | IRes)
26633| {
26634|     return 0;
26635| }
26636|
26637| __declspec( dllexport ) LONG __cdecl InstInstallPsm(
    | LONG *RebootNeeded )
26638| {
26639|     BOOLEAN Need=FALSE;
26640|     ULONG Err;
26641|     Err = Psm_InstallPsm(&Need);
26642|     *RebootNeeded = Need;
26643|     return Err;
26644| }
26645|
26646| __declspec( dllexport ) LONG __cdecl InstUnInstallPsm(
    | LONG *RebootNeeded )
26647| {
26648|     BOOLEAN Need=FALSE;
26649|     ULONG Err;
26650|     Err = Psm_UnInstallPsm(&Need);
26651|     *RebootNeeded = Need;
26652|     return Err;
26653| }
26654|
26655| __declspec( dllexport ) LONG __cdecl
    | InstPsmIsInstalled( LONG *Version, LONG *LoVersion )
26656| {
26657|     tPSM_VersionInfo Ver={0};
26658|     ULONG Err;
26659|
26660|     Ver.Size = sizeof(Ver);
26661|     Err = Psm_IsInstalled(sizeof(Ver),&Ver);
26662|     *Version   = Ver.Version;
26663|     *LoVersion = Ver.LoVersion;
26664|     return Err;
26665| }
26666|
26667| STATIC ULONG CheckSystemForUpgrade()
26668| {
26669|     TCHAR RegKey[255];
26670|     HKEY Key;
26671|     ULONG Err;
26672|     BOOLEAN NeedsRebooting=FALSE;
26673|
26674|
    | _stprintf(RegKey,TEXT("SYSTEM\\CurrentControlSet\\Servic
    | es\\psman4"));
26675|     Err = RegOpenKeyEx(

```

```

26676|     HKEY_LOCAL_MACHINE, // handle of open key
26677|     RegKey, // address of name of subkey to open
26678|     0, // dwOptions
26679|     KEY_ALL_ACCESS, // security access mask
26680|     &Key // address of handle of open key
26681| );
26682| if(!Err) {
26683|     RegCloseKey(Key);
26684|
26685|     // okay key exists, but this is nt5 not 4, so
    | lets assume its an upgrade.
26686|     DLOG(("Upgrade from NT 4 detected!!!
    | Installing Win2k Psm\n"));
26687|
26688|     Err = Psm_InstallPsm(&NeedsRebooting);
26689|
26690|     if((Err==0) || (Err==ERROR_SERVICE_EXISTS)) {
26691|         Err =
    | RegDeleteKey(HKEY_LOCAL_MACHINE,RegKey);
26692|         if(Err) {
26693|             DLOG(("Error %08x deleting NT 4
    | key\n",Err));
26694|         }
26695|     } else {
26696|         DLOG(("Error %08x installing Psm\n",Err));
26697|     }
26698| }
26699| return Err;
26700| }
26701|
26702| ULONG RemoveLogSource( LPTSTR KeyName )
26703| {
26704|     ULONG Err;
26705|     HKEY Key;
26706|
26707|     Err = RegOpenKey(
26708|         HKEY_LOCAL_MACHINE, // handle of open key
26709|         | TEXT("SYSTEM\\CurrentControlSet\\Services\\EventLog\\Sys
    | tem"),
26710|         &Key);
26711|
26712|     if(Err==0) {
26713|         Err = RegDeleteKey( Key, KeyName );
26714|         RegCloseKey(Key);
26715|     }
26716|     return Err;
26717| }
26718|
26719| ULONG AddLogSource( LPTSTR KeyName )

```

```

26720| {
26721|     ULONG Err;
26722|     HKEY Key;
26723|     DWORD Disposition;
26724|     TCHAR RegStr[255];
26725|
26726|
26727|     | _sprintf(RegStr,TEXT("SYSTEM\\CurrentControlSet\\Servic
26728|     | es\\EventLog\\System\\%s"),KeyName);
26729|     Err = RegCreateKeyEx(
26730|         HKEY_LOCAL_MACHINE, // handle of open key
26731|         RegStr, // address of name of subkey to open
26732|         0, // reserved
26733|         NULL, // lpClass
26734|         0, // dwOptions
26735|         KEY_ALL_ACCESS, // security access mask
26736|         NULL, // security attributes
26737|         &Key, // address of handle of open key
26738|         &Disposition
26739|     );
26740|     if(Err==0) {
26741|         // if new key, then add stuff, otherwise we are
26742|         | done
26743|         if(Disposition == REG_CREATED_NEW_KEY) {
26744|             DWORD TypesSupported= 0x7;
26745|
26746|             | _sprintf(RegStr,TEXT("%%SystemRoot%%\\System32\\IoLogMs
26747|             | g.dll;%%SystemRoot%%\\System32\\Drivers\\%s.sys"),KeyNam
26748|             | e);
26749|             Err =
26750|             | RegSetValueEx(Key,TEXT("EventMessageFile"),0,REG_EXPAND_
26751|             | SZ,(BYTE*)RegStr,_tcslen(RegStr));
26752|             Err =
26753|             | RegSetValueEx(Key,TEXT("TypesSupported"),0,REG_DWORD,(BY
26754|             | TE*)&TypesSupported,sizeof(DWORD));
26755|         }
26756|         // close the registry
26757|         RegCloseKey(Key);
26758|     }
26759|     return Err;
26760| }
26761|
26762|
26763|
26764| STATIC void *InitSharedMemory( LPSECURITY_ATTRIBUTES sa
26765|     | )
26766| {
26767|     tSharedMemory *Memory=NULL;
26768|     int InitMemory = FALSE;
26769|     TCHAR MemoryName[100];

```

```

26759|
26760|     SetLastError(0);
26761|
26762|     | _stprintf(MemoryName,"PSM_SHARED_MEMORY_%04x",PSM_LOW_CO
    | MPATIBLE_VERSION);
26763|     MapFileHandle = CreateFileMapping(
26764|         (void*)-1,
26765|         sa,
26766|         PAGE_READWRITE | SEC_COMMIT,
26767|         0,
26768|         sizeof(tSharedMemory),
26769|         MemoryName
26770|     );
26771|
26772|     if(MapFileHandle!=NULL) {
26773|         InitMemory =
    | GetLastError()!=ERROR_ALREADY_EXISTS;
26774|
26775|         Memory = MapViewOfFile(
26776|             MapFileHandle,
26777|             FILE_MAP_WRITE, // read write
26778|             0,
26779|             0,
26780|             0
26781|         );
26782|         if((Memory) && (InitMemory)) {
26783|             memset(Memory,0,sizeof(tSharedMemory));
26784|             Memory->Size = sizeof(tSharedMemory);
26785|         } else {
26786|             if((Memory) &&
    | (Memory->Size!=sizeof(tSharedMemory))) {
26787|                 DLOG(("Error! Shared Memory Allocated
    | size=%d, but already exists as %d
    | size\n",sizeof(tSharedMemory),Memory->Size));
26788|
    | SetLastError(PSM_ERROR_INCOMPATIBLE_DLL);
26789|                 UnmapViewOfFile(Memory);
26790|                 CloseHandle(MapFileHandle);
26791|                 return NULL;
26792|             }
26793|         }
26794|         DLOG(("Shared Memory Allocated
    | size=%d\n",sizeof(tSharedMemory)));
26795|     }
26796|     return Memory;
26797| }
26798|
26799| void DelnitSharedMemory( void *Memory)
26800| {

```

```

26801|  if(Memory) {
26802|      UnmapViewOfFile(Memory);
26803|  }
26804|  if(MapFileHandle!=NULL) {
26805|      CloseHandle(MapFileHandle);
26806|  }
26807|  return;
26808| }
26809|
26810| BOOL WINAPI DllMain(
26811|     HINSTANCE hDllInst,
26812|     DWORD fdwReason,
26813|     LPVOID lpvReserved)
26814| {
26815|     pThreadStorage ThreadStorage;
26816|     BOOL bResult = TRUE;
26817|     SECURITY_ATTRIBUTES sa={0};
26818|     SECURITY_DESCRIPTOR sd={0};
26819|
26820|     // Dispatch this call based on the reason it was
        | called.
26821|     switch (fdwReason)  {
26822|         case DLL_PROCESS_ATTACH:
26823|             // The DLL is being loaded for the first
        | time by a given process.
26824|             // Perform per-process initialization here.
        | If the initialization
26825|             // is successful, return TRUE; if
        | unsuccessful, return FALSE.
26826|
26827|             NtBuildNumber = (GetVersion() >> 16) &
        | 0x3fff;
26828|             {
26829|                 TCHAR BuildStr[80];
26830|                 _sprintf(BuildStr,TEXT("psmlapi: NT
        | build number=%d\n"),NtBuildNumber);
26831|                 OutputDebugString(BuildStr);
26832|             }
26833|
26834|             GetRegistrySettings();
26835|
26836|             pNtQueryVolumeInformation =
        | (tNtQueryVolumeInformation
        | )GetProcAddress(GetModuleHandle(TEXT("ntdll")),TEXT("NtQ
        | ueryVolumeInformationFile"));
26837|
26838|             // check to see if we need to convert to
        | Win2k drivers
26839|             if(NtBuildNumber > 1381) {
26840|                 CheckSystemForUpgrade();

```

```

26841|         pGetVolumePathName =
| (tGetVolumePathName)GetProcAddress(GetModuleHandle(TEXT(
| "kernel32")),TEXT("GetVolumePathNameW"));
26842|         pGetVolumeNameForVolumeMountPoint =
| (tGetVolumeNameForVolumeMountPoint)GetProcAddress(GetMod
| uleHandle(TEXT("kernel32")),TEXT("GetVolumeNameForVolume
| MountPointW"));
26843|         pFindFirstVolume=
| (tFindFirstVolume)GetProcAddress(GetModuleHandle(TEXT("k
| ernel32")),TEXT("FindFirstVolumeW"));
26844|         pFindNextVolume=
| (tFindNextVolume)GetProcAddress(GetModuleHandle(TEXT("ke
| rnel32")),TEXT("FindNextVolumeW"));
26845|         pFindVolumeClose =
| (tFindVolumeClose)GetProcAddress(GetModuleHandle(TEXT("k
| ernel32")),TEXT("FindVolumeClose"));
26846|
26847|         ASSERT(pGetVolumePathName);
26848|
| ASSERT(pGetVolumeNameForVolumeMountPoint);
26849|         ASSERT(pFindFirstVolume);
26850|         ASSERT(pFindNextVolume);
26851|         ASSERT(pFindVolumeClose);
26852|     }
26853|
26854|         DLOG((TEXT("DLL_PROCESS_ATTACH
| %08x\n"),hDllInst));
26855|
26856|         TlsIndex = TlsAlloc();
26857|         // unable to alloc thread local storage
26858|         if(TlsIndex==0xffffffff) {
26859|             DLOG((TEXT("Error %08x obtaining tls
| storage index\n"),GetLastError()));
26860|             return FALSE;
26861|         }
26862|
26863|         // create a null dacl
26864|         InitializeSecurityDescriptor( &sd,
| SECURITY_DESCRIPTOR_REVISION );
26865|         SetSecurityDescriptorDacl(&sd, TRUE, NULL,
| FALSE);
26866|
26867|         sa.nLength = sizeof(SEURITY_ATTRIBUTES);
26868|         sa.bInheritHandle = FALSE;
26869|         sa.lpSecurityDescriptor = &sd;
26870|
26871|         SharedMemoryMutex =
| CreateMutex((LPSECURITY_ATTRIBUTES)&sa,FALSE,TEXT("PSM_S
| haredMemoryMutex"));
26872|         if(!SharedMemoryMutex) {

```

```

26873|         DLOG((TEXT("Error %08x obtaining shared
| mutex\n"),GetLastError()));
26874|         return FALSE;
26875|     }
26876|
26877|     OpenCloseMutex =
| CreateMutex((LPSECURITY_ATTRIBUTES)&sa,FALSE,TEXT("PSM_O
| penCloseMutex"));
26878|     if(!OpenCloseMutex) {
26879|         DLOG((TEXT("Error %08x obtaining open
| close mutex\n"),GetLastError()));
26880|         CloseHandle(SharedMemoryMutex);
26881|         return FALSE;
26882|     }
26883|     SharedMemory =
| InitSharedMemory((LPSECURITY_ATTRIBUTES)&sa);
26884|     if(!SharedMemory) {
26885|         DLOG((TEXT("Error %08x obtaining shared
| memory address\n"),GetLastError()));
26886|         CloseHandle(OpenCloseMutex);
26887|         CloseHandle(SharedMemoryMutex);
26888|         return FALSE;
26889|     }
26890|
26891|     // fall through for each initial thread
26892|     case DLL_THREAD_ATTACH:
26893|         // A thread is being created in a process
| that has already loaded
26894|         // this DLL. Perform any per-thread
| initialization here. The
26895|         // return value is ignored.
26896|         DLOG((TEXT("DLL_THREAD_ATTACH
| %08x\n"),hDllInst));
26897|
26898|         ThreadStorage =
| LocalAlloc(LPTR,sizeof(tThreadStorage));
26899|         if(!ThreadStorage)
26900|             return FALSE;
26901|
26902|         TlsSetValue(TlsIndex,ThreadStorage);
26903|
26904|         InitForNewThread();
26905|
26906|         break;
26907|
26908|     case DLL_PROCESS_DETACH:
26909|         if ( FNVFS_ListOfVolumeNames != NULL ) {
26910|             LocalFree (FNVFS_ListOfVolumeNames);
26911|             FNVFS_ListOfVolumeNames = NULL;
26912|         }

```

```

26913|         FNVFS_ListOfVolumeNamesSizeInBytes = 0;
26914|         // fall through... no 'break'
26915|
26916|         case DLL_THREAD_DETACH:
26917|
26918|             // A thread is exiting cleanly in a process
                | that has already
26919|             // loaded this DLL. Perform any per-thread
                | clean up here. The
26920|             // return value is ignored.
26921|             DLOG((TEXT("DLL_THREAD_DETACH
                | %08x\n"),hDllInst));
26922|
26923|             ThreadStorage = GetThreadStorage();
26924|
26925|             if(!ThreadStorage->Persistent) {
26926|                 // close all outstanding open psms
26927|                 while(ThreadStorage->NumOpens>0) {
26928|                     DLOG((TEXT("Closing psm left open
                | by app (%d)\n"),ThreadStorage->NumOpens));
26929|                     Psm_DestroySnapShot(NULL);
26930|                 }
26931|             }
26932|
26933|             if
                | (ThreadStorage->PSManEvent!=INVALID_HANDLE_VALUE) {
26934|                 CloseHandle(ThreadStorage->PSManEvent);
26935|                 ThreadStorage->PSManEvent =
                | INVALID_HANDLE_VALUE;
26936|             }
26937|
26938|             | if(ThreadStorage->PSManHandle!=INVALID_HANDLE_VALUE) {
26939|                 PSMI_CloseManager(
                | ThreadStorage->PSManHandle );
26940|                 ThreadStorage->PSManHandle =
                | INVALID_HANDLE_VALUE;
26941|             }
26942|
26943| #ifdef _DEBUG
26944|             if(ThreadStorage->DebugLogFile) {
26945|                 | CloseHandle(ThreadStorage->DebugLogFile);
26946|                 ThreadStorage->DebugLogFile =
                | INVALID_HANDLE_VALUE;
26947|             }
26948| #endif
26949|             LocalFree((HLOCAL)ThreadStorage);
26950|
26951|             if(fdwReason==DLL_THREAD_DETACH)

```



```

26952|         break;
26953|
26954|         // The DLL is being unloaded by a given
        | process. Do any
26955|         // per-process clean up here, such as
        | undoing what was done in
26956|         // DLL_PROCESS_ATTACH. The return value is
        | ignored.
26957|         // Unmap any shared memory from the
        | process's address space.
26958|         DLOG((TEXT("DLL_PROCESS_DETACH
        | %08x\n"),hDllInst));
26959|
26960|
26961|         TlsFree(TlsIndex);
26962|         CloseHandle(SharedMemoryMutex);
26963|         CloseHandle(OpenCloseMutex);
26964|         DelInitSharedMemory(SharedMemory);
26965|         break;
26966|     }
26967|     return (bResult);
26968| }
26969|
26970| //-----
        | -----
26971|
26972| DLLEXPORT PSMSTATUS PSMAPI Psm_DeletePsmFileW (
26973|     const WCHAR *PsmFilePath )        //
        | L"??\Volume{...}\...\xxx.psm"
26974| {
26975|     NTSTATUS Status = 0;
26976|     UNICODE_STRING Uni = {0};
26977|     OBJECT_ATTRIBUTES ObjectAttributes = {0};
26978|     ULONG Err = 0;
26979|     HANDLE FileHandle = INVALID_HANDLE_VALUE;
26980|     IO_STATUS_BLOCK IoStatus = {0};
26981|
26982|     RtlInitUnicodeString ( &Uni, PsmFilePath );
26983|
26984|     InitializeObjectAttributes (
26985|         &ObjectAttributes,
26986|         &Uni,
26987|         OBJ_CASE_INSENSITIVE,
26988|         NULL,
26989|         NULL );
26990|
26991|     Status = NtCreateFile (
26992|         &FileHandle,
26993|         STANDARD_RIGHTS_ALL,        // desired
        | access

```

```

26994|    &ObjectAttributes,    // object
    | attributes
26995|    &IoStatus,
26996|    NULL,    // alloc size
26997|    FILE_ATTRIBUTE_NORMAL,    // file attributes
26998|    0,    // share access
26999|    FILE_OPEN,    // create disposition
27000|    FILE_SYNCHRONOUS_IO_NONALERT,    //
    | create options
27001|    NULL, // eabuffer
27002|    0 ); // ealength
27003|
27004|    if ( Status == 0 ) {
27005|        FILE_DISPOSITION_INFORMATION Del = {0};
27006|        Del.DeleteFile = TRUE;
27007|        Status = NtSetInformationFile (
27008|            FileHandle,
27009|            &IoStatus,
27010|            &Del,
27011|            sizeof(Del),
27012|            FileDispositionInformation );
27013|        if ( Status != 0 ) {
27014|            DLOG((TEXT("Psm_DeletePsmFile:
    | NtSetInformationFile returned %08x\n"),Status));
27015|        }
27016|    } else {
27017|        DLOG((TEXT("Psm_DeletePsmFile: NtCreateFile
    | returned %08x\n"),Status));
27018|    }
27019|
27020|    return Status;
27021| }
27022|
27023| DLLEXPORT PSMSTATUS PSMAPI Psm_DeletePsmFileA (
27024|    const CHAR *PsmFilePath )    //
    | L"\\??\\Volume{...}\\...\\xxx.psm"
27025| {
27026|    WCHAR PsmFilePathW[1025];
27027|    OemToCharW(PsmFilePath,PsmFilePathW);
27028|    return Psm_DeletePsmFileW(PsmFilePathW);
27029| }
27030|
27031| //-----
    | -----
27032|
27033|
27034|
27035| File Listing: RESOURCE.h
27036|
27037| //{NO_DEPENDENCIES}

```

```

27038| // Microsoft Developer Studio generated include file.
27039| // Used by SBPSMAN.RC
27040| //
27041| #define VER_PRODUCTBUILD          3
27042| #define VER_PRODUCTVERSION_W      0x0101
27043|
27044| // Next default values for new objects
27045| //
27046| #ifdef APSTUDIO_INVOKED
27047| #ifndef APSTUDIO_READONLY_SYMBOLS
27048| #define _APS_NO_MFC                1
27049| #define _APS_NEXT_RESOURCE_VALUE    101
27050| #define _APS_NEXT_COMMAND_VALUE     40001
27051| #define _APS_NEXT_CONTROL_VALUE     1000
27052| #define _APS_NEXT_SYMED_VALUE      101
27053| #endif
27054| #endif
27055|
27056|
27057|
27058| File Listing: safemem.c
27059|
27060| #include <stdio.h>
27061| #include <stdlib.h>
27062| #include <string.h>
27063| #include <windows.h>
27064| #include <windowsx.h>
27065| #include <winioctl.h>
27066| #include <tchar.h>
27067| #include <conio.h>
27068| // #include <crtdbg.h>          // _ASSERTE
27069| #include <assert.h>
27070| #define _ASSERTE assert
27071|
27072| #include "safemem.h"
27073|
27074| ULONG GetSystemPageSize()
27075| {
27076|     SYSTEM_INFO SI={0};
27077|     GetSystemInfo( &SI );
27078|     return SI.dwPageSize;
27079| }
27080|
27081| #define ULONGLONG unsigned __int64
27082| // variables for memory tracking
27083| ULONGLONG  TotalMemoryAllocated      = 0;
27084| ULONGLONG  TotalMemoryAllocations    = 0;
27085| ULONGLONG  LargestMemoryAllocation   = 0;
27086| ULONGLONG  TotalMemoryAllocatedAtOnce = 0;
27087| ULONGLONG  TotalMemoryAllocationsAtOnce = 0;

```

```

27088|
27089| /*
27090|   This routine will alloc Size bytes, and protect the
      | memory after it so buffer
27091|   overflowing will be caught during debugging.
27092| */
27093| void *SAFE_Alloc( ULONG Size )
27094| {
27095|   ULONG NumPages = (Size+(GetSystemPageSize()-1)) /
      | GetSystemPageSize();
27096|   ULONG Offset   = (Size) % GetSystemPageSize() ?
      | (Size) % GetSystemPageSize() : GetSystemPageSize();
27097|   BYTE *Buffer   = VirtualAlloc( NULL,
      | (NumPages*GetSystemPageSize())+(GetSystemPageSize()*2),
      | MEM_RESERVE | MEM_COMMIT, PAGE_NOACCESS );
27098|   ULONG OldAccess;
27099|
27100|   if(Buffer) {
27101|       // store size pointer, we need it for free
27102|
      | VirtualProtect(Buffer,GetSystemPageSize(),PAGE_READWRITE
      | ,&OldAccess);
27103|       *((ULONG*)(Buffer)) =
      | NumPages*GetSystemPageSize();
27104|       *((ULONG*)(Buffer+4)) = Size;
27105|
      | VirtualProtect(Buffer,GetSystemPageSize(),PAGE_NOACCESS,
      | &OldAccess);
27106|
27107|       // test
27108|       /**((ULONG*)(Buffer)) = 1;
27109|
27110|       // commit user buffer, only can change accesses
      | in page increments
27111|
      | VirtualProtect(Buffer+GetSystemPageSize(),Size,PAGE_READ
      | WRITE,&OldAccess);
27112|
27113|       // record some statistics
27114|       TotalMemoryAllocated+=Size;
27115|
      | if(TotalMemoryAllocated>TotalMemoryAllocatedAtOnce)
27116|       TotalMemoryAllocatedAtOnce =
      | TotalMemoryAllocated;
27117|
27118|       TotalMemoryAllocations++;
27119|
      | if(TotalMemoryAllocations>TotalMemoryAllocationsAtOnce)
27120|       TotalMemoryAllocationsAtOnce =
      | TotalMemoryAllocations;

```

```

27121|
27122|     if(Size>LargestMemoryAllocation)
27123|         LargestMemoryAllocation = Size;
27124|
27125|     // return where buffer overflow will generate
        | access violation
27126| #if 0
27127|
        | _tprintf(TEXT("          Alloc:
        | %04x \n"),
27128|                (Size)
27129|                );
27130| #endif
27131|     return
        | Buffer+GetSystemPageSize()+(GetSystemPageSize()-Offset);
27132| } else {
27133|     // _RPT0( _CRT_WARN, "SAFE_Alloc: NULL pointer
        | being returned.\n");
27134|     return NULL;
27135| }
27136| }
27137|
27138| /*
27139| This frees a pointer allocated with NTFS_SafeAlloc
27140| */
27141| void SAFE_Free( PVOID Buffer )
27142| {
27143|     if(Buffer) {
27144|         ULONG PageSize;
27145|         ULONG Size;
27146|         ULONG OldAccess;
27147|         // get the normallized buffer address
27148|         BYTE *SystemBuffer =
            | (BYTE*)((((ULONG)((((BYTE*)Buffer)-GetSystemPageSize())) /
            | GetSystemPageSize()) * GetSystemPageSize());
27149|
27150|         // get size of buffer
27151|
            | VirtualProtect(SystemBuffer,GetSystemPageSize(),PAGE_REA
            | DONLY,&OldAccess);
27152|         PageSize = ((ULONG*)SystemBuffer)[0];
27153|         Size     = ((ULONG*)SystemBuffer)[1];
27154|
27155|         // and free it
27156|         if(!VirtualFree( SystemBuffer,
            | PageSize+(GetSystemPageSize()*2), MEM_DECOMMIT )) {
27157|             // _RPT1( _CRT_WARN, "SAFE_Free:
            | VirtualFree decommit failed %08x\n",GetLastError());
27158|         }
27159|         if(!VirtualFree( SystemBuffer, 0, MEM_RELEASE

```

```

    | )) {
27160|         // _RPT1( _CRT_WARN, "SAFE_Free:
    | VirtualFree release failed %08x\n", GetLastError());
27161|     }
27162|
27163|     // Stats
27164|     TotalMemoryAllocated-=Size;
27165|     TotalMemoryAllocations--;
27166| #if 0
27167|
    | _tprintf(TEXT("                Freed:
    | %04x \n"),
27168|                (Size)
27169|                );
27170| #endif
27171| } else {
27172|     // _RPT0( _CRT_WARN, "SAFE_Free: NULL pointer
    | passed to SAFE_Free.\n");
27173| }
27174| return;
27175| }
27176|
27177| /*
27178|  Protects a buffer allocated with NTFS_SafeAlloc,
    | any access to the buffer will result in
27179|  an access violation (Exception) being generated
27180|  */
27181| void SAFE_NoAccess( PVOID Buffer )
27182| {
27183|     if(Buffer) {
27184|         ULONG PageSize;
27185|         ULONG OldAccess;
27186|         // get the normallized buffer address
27187|         BYTE *SystemBuffer =
            | (BYTE*)((ULONG)((BYTE*)Buffer)-GetSystemPageSize()) /
            | GetSystemPageSize() * GetSystemPageSize());
27188|
27189|         // get size of buffer
27190|
            | VirtualProtect(SystemBuffer,GetSystemPageSize(),PAGE_REA
            | DONLY,&OldAccess);
27191|         PageSize = ((ULONG*)SystemBuffer)[0];
27192|
            | VirtualProtect(SystemBuffer,GetSystemPageSize(),PAGE_NOA
            | CCESS,&OldAccess);
27193|
27194|         // commit user buffer, only can change accesses
            | in page increments
27195|
            | VirtualProtect(SystemBuffer+GetSystemPageSize(),PageSize

```

```

    | ,PAGE_NOACCESS,&OldAccess);
27196| }
27197| return;
27198| }
27199|
27200| /*
27201|  Makes the buffer allocated with NTFS_SafeAlloc read
    | only, writes to the buffer
27202|  will generate an access violation
27203| */
27204| void SAFE_ReadOnly( PVOID Buffer )
27205| {
27206|     if(Buffer) {
27207|         ULONG PageSize;
27208|         ULONG OldAccess;
27209|         // get the normallized buffer address
27210|         BYTE *SystemBuffer =
            | (BYTE*)((ULONG)((BYTE*)Buffer)-GetSystemPageSize()) /
            | GetSystemPageSize() * GetSystemPageSize());
27211|
27212|         // get size of buffer
27213|
            | VirtualProtect(SystemBuffer,GetSystemPageSize(),PAGE_REA
            | DONLY,&OldAccess);
27214|         PageSize = ((ULONG*)SystemBuffer)[0];
27215|
            | VirtualProtect(SystemBuffer,GetSystemPageSize(),PAGE_NOA
            | CCESS,&OldAccess);
27216|
27217|         // commit user buffer, only can change accesses
            | in page increments
27218|
            | VirtualProtect(SystemBuffer+GetSystemPageSize(),PageSize
            | ,PAGE_READONLY,&OldAccess);
27219|     }
27220|     return;
27221| }
27222|
27223| /*
27224|  Makes the buffer readable and writeable (normal)
27225| */
27226| void SAFE_ReadWrite( PVOID Buffer )
27227| {
27228|     if(Buffer) {
27229|         ULONG PageSize;
27230|         ULONG OldAccess;
27231|         // get the normallized buffer address
27232|         BYTE *SystemBuffer =
            | (BYTE*)((ULONG)((BYTE*)Buffer)-GetSystemPageSize()) /
            | GetSystemPageSize() * GetSystemPageSize());

```

```

27233|
27234|    // get size of buffer
27235|
27236|    | VirtualProtect(SystemBuffer,GetSystemPageSize(),PAGE_REA
27237|    | DONLY,&OldAccess);
27238|    PageSize = ((ULONG*)SystemBuffer)[0];
27239|
27240|    | VirtualProtect(SystemBuffer,GetSystemPageSize(),PAGE_NOA
27241|    | CCESS,&OldAccess);
27242|
27243|    // commit user buffer, only can change accesses
27244|    | in page increments
27245|
27246|    | VirtualProtect(SystemBuffer+GetSystemPageSize(),PageSize
27247|    | ,PAGE_READWRITE,&OldAccess);
27248|    }
27249|    return;
27250| }
27251|
27252| void SAFE_LogMemoryUsage()
27253| {
27254|    _tprintf(TEXT("Total Memory Allocated      :
27255|    | %!64d Bytes\n"),TotalMemoryAllocated);
27256|    _tprintf(TEXT("Total Memory Allocations      :
27257|    | %!64d\n"),TotalMemoryAllocations);
27258|    _tprintf(TEXT("Largest Memory Allocation      :
27259|    | %!64d Bytes\n"),LargestMemoryAllocation);
27260|    _tprintf(TEXT("Total Memory Allocated At Once :
27261|    | %!64d Bytes\n"),TotalMemoryAllocatedAtOnce);
27262|    _tprintf(TEXT("Total Memory Allocations At Once:
27263|    | %!64d\n"),TotalMemoryAllocationsAtOnce);
27264| }
27265|
27266| void SAFE_VerifyMem()
27267| {
27268|    _ASSERTE(TotalMemoryAllocated==0);
27269| }
27270|
27271|
27272|
27273|
27274| File Listing: safemem.h
27275|
27276| ULONG GetSystemPageSize();
27277| void *SAFE_Alloc( ULONG Size );
27278| void SAFE_Free( PVOID Buffer );
27279| void SAFE_NoAccess( PVOID Buffer );
27280| void SAFE_ReadOnly( PVOID Buffer );
27281| void SAFE_ReadWrite( PVOID Buffer );
27282| void SAFE_LogMemoryUsage();
27283| void SAFE_VerifyMem();

```



```

27271|
27272|
27273|
27274| File Listing: SERVICE.c
27275|
27276| #include <stdio.h>
27277| #include <stdlib.h>
27278| #include <windows.h>
27279| #include <windowsx.h>
27280| #include <string.h>
27281| #include <direct.h>
27282| #include <memory.h>
27283| #include <tchar.h>
27284| #include <winerror.h>
27285|
27286| //
27287| // FUNCTION: CmdInstallService()
27288| //
27289| // PURPOSE: Installs the service
27290| //
27291| // PARAMETERS:
27292| //   none
27293| //
27294| // RETURN VALUE:
27295| //   none
27296| //
27297| // COMMENTS:
27298| //
27299|
27300| int Svc_InstallService(
27301|     LPTSTR FullPathName,
27302|     LPTSTR ServiceName,
27303|     LPTSTR DisplayName,
27304|     LPTSTR Dependencies,
27305|     LPTSTR Group,
27306|     int ServiceType,
27307|     int StartType,
27308|     int Tag
27309| )
27310| {
27311|     SC_HANDLE schService;
27312|     SC_HANDLE schSCManager;
27313|     int Err;
27314|
27315|     schSCManager = OpenSCManager(
27316|         NULL, //
27317|         | machine (NULL == local)
27318|         NULL, //
27319|         | database (NULL == default)
27320|         SC_MANAGER_ALL_ACCESS //

```

```

    | access required
27319|         );
27320|
27321|     if ( schSCManager ) {
27322|         schService = CreateService(
27323|             schSCManager,          // SCManager
    | database
27324|             ServiceName,          // name of service
27325|             DisplayName, // name to display
27326|             SERVICE_ALL_ACCESS,    // desired
    | access
27327|             ServiceType, // service type
27328|             StartType,    // start type
27329|             SERVICE_ERROR_NORMAL, // error
    | control type
27330|             FullPathName,         //
    | service's binary
27331|             Group,                // no load
    | ordering group
27332|             &Tag,                // no tag
    | identifier
27333|             Dependencies,         // dependencies
27334|             NULL,                // LocalSystem
    | account
27335|             NULL);              // no password
27336|
27337|     if ( schService ) {
27338|         Err = 0;
27339|         CloseServiceHandle(schService);
27340|     } else {
27341|         Err = GetLastError();
27342|     }
27343|
27344|     CloseServiceHandle(schSCManager);
27345| } else
27346|     Err = GetLastError();
27347|
27348| return Err;
27349| }
27350|
27351|
27352|
27353| //
27354| // FUNCTION: CmdRemoveService()
27355| //
27356| // PURPOSE: Stops and removes the service
27357| //
27358| // PARAMETERS:
27359| //     none
27360| //

```

```

27361| // RETURN VALUE:
27362| //  none
27363| //
27364| // COMMENTS:
27365| //
27366| int Svc_RemoveService( LPTSTR ServiceName )
27367| {
27368|     SC_HANDLE  schService;
27369|     SC_HANDLE  schSCManager;
27370|     SERVICE_STATUS  ssStatus;    // current status
        | of the service
27371|     int Err=0;
27372|
27373|     schSCManager = OpenSCManager(
27374|         NULL,          //
        | machine (NULL == local)
27375|         NULL,          //
        | database (NULL == default)
27376|         SC_MANAGER_ALL_ACCESS  //
        | access required
27377|     );
27378|
27379|     if ( schSCManager ) {
27380|         schService = OpenService(schSCManager,
        | ServiceName, SERVICE_ALL_ACCESS);
27381|
27382|         if (schService) {
27383|             // try to stop the service
27384|             if ( ControlService( schService,
        | SERVICE_CONTROL_STOP, &ssStatus ) ) {
27385|                 Sleep( 1000 );
27386|
27387|                 while( QueryServiceStatus( schService,
        | &ssStatus ) ) {
27388|                     if ( ssStatus.dwCurrentState ==
        | SERVICE_STOP_PENDING ) {
27389|                         Sleep( 1000 );
27390|                     } else
27391|                         break;
27392|                 }
27393|
27394|                 if ( ssStatus.dwCurrentState ==
        | SERVICE_STOPPED )
27395|                     Err = 0;
27396|                 else
27397|                     Err = GetLastError();
27398|
27399|             }
27400|
27401|             // now remove the service

```

```

27402|         if( DeleteService(schService) )
27403|             Err = 0;
27404|         else
27405|             Err = GetLastError();
27406|
27407|
27408|         CloseServiceHandle(schService);
27409|     } else
27410|         Err = GetLastError();
27411|
27412|         CloseServiceHandle(schSCManager);
27413|     } else
27414|         Err = GetLastError();
27415|
27416|     return Err;
27417| }
27418|
27419| //
27420| // FUNCTION: Svc_StopService()
27421| //
27422| // PURPOSE: Stops the service
27423| //
27424| // PARAMETERS:
27425| //     none
27426| //
27427| // RETURN VALUE:
27428| //     none
27429| //
27430| // COMMENTS:
27431| //
27432| int Svc_StopService( LPTSTR ServiceName )
27433| {
27434|     SC_HANDLE  schService;
27435|     SC_HANDLE  schSCManager;
27436|     SERVICE_STATUS  ssStatus;    // current status
        | of the service
27437|     int Err=0;
27438|
27439|     schSCManager = OpenSCManager(
27440|         NULL,          //
        | machine (NULL == local)
27441|         NULL,          //
        | database (NULL == default)
27442|         SC_MANAGER_ALL_ACCESS  //
        | access required
27443|     );
27444|
27445|     if ( schSCManager ) {
27446|         schService = OpenService(schSCManager,
        | ServiceName, SERVICE_ALL_ACCESS);

```

```

27447|
27448|     if (schService) {
27449|         // try to stop the service
27450|         if ( ControlService( schService,
27451|             | SERVICE_CONTROL_STOP, &ssStatus ) ) {
27452|             Sleep( 1000 );
27453|             while( QueryServiceStatus( schService,
27454|                 | &ssStatus ) ) {
27455|                 if ( ssStatus.dwCurrentState ==
27456|                     | SERVICE_STOP_PENDING ) {
27457|                     Sleep( 1000 );
27458|                 } else
27459|                 {
27460|                     break;
27461|                 }
27462|                 if ( ssStatus.dwCurrentState ==
27463|                     | SERVICE_STOPPED )
27464|                     Err = 0;
27465|                 else
27466|                     Err = GetLastError();
27467|             }
27468|             CloseServiceHandle(schService);
27469|         } else
27470|             Err = GetLastError();
27471|         CloseServiceHandle(schSCManager);
27472|     } else
27473|         Err = GetLastError();
27474|
27475|     return Err;
27476| }
27477|
27478| //
27479| // FUNCTION: CmdStartService()
27480| //
27481| // PURPOSE: Stops and removes the service
27482| //
27483| // PARAMETERS:
27484| //     none
27485| //
27486| // RETURN VALUE:
27487| //     none
27488| //
27489| // COMMENTS:
27490| //
27491| int Svc_StartService( LPTSTR ServiceName )
27492| {

```

```

27493| SC_HANDLE schService;
27494| SC_HANDLE schSCManager;
27495| int Err=0;
27496| TCHAR *p=NULL;
27497| TCHAR *(Argv[10]) = {0};
27498| LONG Argc=0;
27499| TCHAR RegKey[128];
27500| HKEY Key;
27501| LONG DataSize;
27502| TCHAR ImagePath[128];
27503|
27504| _tcsncpy( RegKey,
    | TEXT("SYSTEM\\CurrentControlSet\\Services\\") );
27505| _tcsncpy( RegKey, ServiceName );
27506|
27507| // open the registry
27508| Err = RegOpenKeyEx(
27509|     HKEY_LOCAL_MACHINE, // handle of open key
27510|     RegKey, // address of name of subkey to open
27511|     0, // reserved
27512|     KEY_READ, // security access mask
27513|     &Key // address of handle of open key
27514| );
27515| if(Err==0) {
27516|     DataSize = _MAX_PATH;
27517|     Err = RegQueryValueEx(
27518|         Key, // handle of key to set
27519|         TEXT("ImagePath"), // address of name of
    | value to query
27520|         NULL, // reserved
27521|         NULL, // value type
27522|         (char*)&ImagePath, // address of data
    | buffer
27523|         &DataSize // address of data buffer size
27524|     );
27525|
27526|     // close the registry
27527|     RegCloseKey(Key);
27528| }
27529|
27530| Argv[0] = ImagePath;
27531| Argv[1] = NULL;
27532| Argc = 1;
27533|
27534| schSCManager = OpenSCManager(
27535|     NULL, //
    | machine (NULL == local)
27536|     NULL, //
    | database (NULL == default)
27537|     SC_MANAGER_ALL_ACCESS //

```

```

    | access required
27538|         );
27539|
27540|     if ( schSCManager ) {
27541|         schService = OpenService(schSCManager,
    | ServiceName, SERVICE_ALL_ACCESS);
27542|
27543|         if (schService) {
27544|             // try to start the service
27545|             if(StartService(
27546|                 schService, // handle of service
27547|                 Argc, // number of arguments
27548|                 Argv // address of array of argument
    | string pointers
27549|             )==FALSE) {
27550|                 Err = GetLastError();
27551|             }
27552|             CloseServiceHandle(schService);
27553|         } else
27554|             Err = GetLastError();
27555|
27556|         CloseServiceHandle(schSCManager);
27557|     } else
27558|         Err = GetLastError();
27559|
27560|     return Err;
27561| }
27562|
27563|
27564| //
27565| // FUNCTION: Svc_CheckServiceExists()
27566| //
27567| // PURPOSE: Checks to see if the service is already
    | installed
27568| //
27569| // PARAMETERS:
27570| //     none
27571| //
27572| // RETURN VALUE:
27573| //     0 if installed, otherwise error code
27574| //
27575| // COMMENTS:
27576| //
27577| int Svc_CheckServiceExists( LPTSTR ServiceName )
27578| {
27579|     SC_HANDLE schService;
27580|     SC_HANDLE schSCManager;
27581|     int Err=0;
27582|
27583|     schSCManager = OpenSCManager(

```

```

27584|          NULL,          //
      | machine (NULL == local)
27585|          NULL,          //
      | database (NULL == default)
27586|          SC_MANAGER_ALL_ACCESS //
      | access required
27587|          );
27588|
27589|  if ( schSCManager ) {
27590|      schService = OpenService(schSCManager,
      | ServiceName, SERVICE_ALL_ACCESS);
27591|
27592|      if (schService) {
27593|          Err = 0;
27594|          CloseServiceHandle(schService);
27595|      } else
27596|          Err = GetLastError();
27597|
27598|      CloseServiceHandle(schSCManager);
27599|  } else
27600|      Err = GetLastError();
27601|
27602|  return Err;
27603| }
27604|
27605|
27606| //
27607| // FUNCTION: CmdChangeServiceStart()
27608| //
27609| // PURPOSE: Changes service start type
27610| //
27611| // PARAMETERS:
27612| //  none
27613| //
27614| // RETURN VALUE:
27615| //  none
27616| //
27617| // COMMENTS:
27618| //
27619| int Svc_ChangeServiceStart( LPTSTR ServiceName, LONG
      | StartType )
27620| {
27621|     SC_HANDLE schService;
27622|     SC_HANDLE schSCManager;
27623|     int Err=0;
27624|
27625|     schSCManager = OpenSCManager(
27626|         NULL,          //
      | machine (NULL == local)
27627|         NULL,          //

```



```

| database (NULL == default)
27628|          SC_MANAGER_ALL_ACCESS  //
| access required
27629|          );
27630|
27631|  if ( schSCManager ) {
27632|      schService = OpenService(schSCManager,
| ServiceName, SERVICE_ALL_ACCESS);
27633|
27634|      if (schService) {
27635|          if (ChangeServiceConfig(
27636|              schService,      // handle to
| service
27637|              SERVICE_NO_CHANGE, // type of service
27638|              StartType,      // when to start
| service
27639|              SERVICE_NO_CHANGE, // severity if
| service fails to start
27640|              NULL,          // pointer to
| service binary file name
27641|              NULL,          // pointer to load
| ordering group name
27642|              NULL,          // pointer to
| variable to get tag identifier
27643|              NULL,          // pointer to array
| of dependency names
27644|              NULL,          // pointer to
| account name of service
27645|              NULL,          // pointer to
| password for service account
27646|              NULL          // pointer to
| display name
27647|          ))
27648|          Err = 0;
27649|      else
27650|          Err = GetLastError();
27651|
27652|          CloseServiceHandle(schService);
27653|      } else
27654|          Err = GetLastError();
27655|
27656|          CloseServiceHandle(schSCManager);
27657|      } else
27658|          Err = GetLastError();
27659|
27660|  return Err;
27661| }
27662|
27663|
27664|

```

```

27665| File Listing: SERVICE.h
27666|
27667| int Svc_InstallService(
27668|         LPTSTR FullPathName,
27669|         LPTSTR ServiceName,
27670|         LPTSTR DisplayName,
27671|         LPTSTR Dependencies,
27672|         LPTSTR Group,
27673|         int ServiceType,
27674|         int StartType,
27675|         int Tag
27676|     );
27677| int Svc_RemoveService( LPTSTR ServiceName );
27678| int Svc_StopService( LPTSTR ServiceName );
27679| int Svc_StartService( LPTSTR ServiceName );
27680| int Svc_CheckServiceExists( LPTSTR ServiceName );
27681| int Svc_ChangeServiceStart( LPTSTR ServiceName, LONG
    | StartType );
27682|
27683|
27684|
27685| File Listing: SETUP4.c
27686|
27687| #include <windows.h>
27688| #include <stdio.h>
27689| #include <malloc.h>
27690|
27691|
27692| // for all of the _t stuff (to allow compiling for both
    | Unicode/Ansi)
27693| #include <tchar.h>
27694|
27695| #include "service.h"
27696|
27697| extern ULONG AddLogSource( LPTSTR KeyName );
27698| extern ULONG RemoveLogSource( LPTSTR KeyName );
27699|
27700|
27701| ULONG SetUpForWinNT( BOOLEAN *RebootNeeded )
27702| {
27703|     ULONG Err;
27704|
27705|     *RebootNeeded = TRUE;
27706|
27707|     AddLogSource(TEXT("psman4"));
27708|
27709|     // install device into registry
27710|     Err = Svc_InstallService(
27711|         TEXT("System32\\DRIVERS\\psman4.sys"),
27712|         TEXT("psman4"),

```

```

27713|     TEXT("Persistent Storage Manager"),
27714|     NULL,
27715|     TEXT("Filter"),
27716|     1,
27717|     0,
27718|     5
27719| );
27720|
27721| if(Err == ERROR_SERVICE_EXISTS) {
27722|     Err = 0;
27723| }
27724| return Err;
27725| }
27726|
27727| ULONG UnSetUpForWinNT( BOOLEAN *RebootNeeded )
27728| {
27729|     *RebootNeeded = TRUE;
27730|     RemoveLogSource(TEXT("psman4"));
27731|     return Svc_RemoveService(TEXT("psman4"));
27732| }
27733|
27734|
27735|
27736| File Listing: SETUP4.h
27737|
27738| ULONG SetUpForWinNT( BOOLEAN *RebootNeeded );
27739| ULONG UnSetUpForWinNT( BOOLEAN *RebootNeeded );
27740|
27741|
27742|
27743| File Listing: SETUP5.c
27744|
27745| // define to install/uninstall psm without rebooting.
27746| // 10-10-99 not working yet..
27747| //#define DO_NT5_SETUP
27748|
27749| #include <windows.h>
27750| #include <stdio.h>
27751| #include <malloc.h>
27752|
27753| // !!!! CAVEAT !!! make sure the Win2k ddk is first in
    | path !!!
27754| // defines GUID
27755| #include <initguid.h>
27756| #include <devguid.h>
27757|
27758| // the SetupDiXXX api (from the DDK)
27759| #include <setupapi.h>
27760|
27761| // defines guids for device classes (DiskClassGuid,

```

```

| etc)
27762| #include <devioctl.h>
27763| #include <ntddstor.h>
27764|
27765| // for all of the _t stuff (to allow compiling for both
| Unicode/Ansi)
27766| #include <tchar.h>
27767|
27768| #include "service.h"
27769| #include "dlog.h"
27770|
27771| #ifdef DEBUG
27772| #define STATIC
27773| #else
27774| #define STATIC static
27775| #endif
27776|
27777| extern ULONG AddLogSource( LPTSTR KeyName );
27778| extern ULONG RemoveLogSource( LPTSTR KeyName );
27779|
27780| #if DO_NT5_SETUP
27781|
27782| typedef struct _sW2kSetupStuff {
27783|     BOOL (WINAPI *SetupDiDestroyDeviceInfoList)( IN
| HDEVINFO DeviceInfoSet );
27784|     BOOL (WINAPI *SetupDiEnumDeviceInfo)( IN HDEVINFO
| DeviceInfoSet, IN DWORD MemberIndex, OUT
| PSP_DEVINFO_DATA DeviceInfoData );
27785|     HDEVINFO (WINAPI *SetupDiGetClassDevs)( IN CONST
| GUID *ClassGuid, IN PCTSTR Enumerator, IN HWND
| hwndParent, IN DWORD Flags );
27786|     BOOL (WINAPI *SetupDiGetDeviceInstallParams)( IN
| HDEVINFO DeviceInfoSet, IN PSP_DEVINFO_DATA
| DeviceInfoData, OUT PSP_DEVINSTALL_PARAMS
| DeviceInstallParams );
27787|     BOOL (WINAPI *SetupDiCallClassInstaller)( IN
| DI_FUNCTION InstallFunction, IN HDEVINFO DeviceInfoSet,
| IN PSP_DEVINFO_DATA DeviceInfoData );
27788|     BOOL (WINAPI *SetupDiSetClassInstallParams)( IN
| HDEVINFO DeviceInfoSet, IN PSP_DEVINFO_DATA
| DeviceInfoData, IN PSP_CLASSINSTALL_HEADER
| ClassInstallParams, IN DWORD ClassInstallParamsSize );
27789|     BOOL (WINAPI *SetupDiGetDeviceRegistryProperty)( IN
| HDEVINFO DeviceInfoSet, IN PSP_DEVINFO_DATA
| DeviceInfoData, IN DWORD Property, OUT PDWORD
| PropertyRegDataType, OUT PBYTE PropertyBuffer, IN DWORD
| PropertyBufferSize, OUT PDWORD RequiredSize );
27790| } tW2kSetup, *pW2kSetup;
27791|
27792| pW2kSetup W2k=NULL;

```

```

27793| HMODULE SetupDll=INVALID_HANDLE_VALUE;
27794|
27795|
27796| STATIC PBYTE
27797| GetDeviceRegistryProperty(
27798|     IN HDEVINFO DeviceInfoSet,
27799|     IN PSP_DEVINFO_DATA DeviceInfoData,
27800|     IN DWORD Property,
27801|     OUT PDWORD PropertyRegDataType
27802| );
27803|
27804| STATIC BOOLEAN
27805| RestartDevice(
27806|     IN HDEVINFO DeviceInfoSet,
27807|     IN OUT PSP_DEVINFO_DATA DeviceInfoData
27808| );
27809| #endif
27810|
27811| STATIC BOOLEAN
27812| PrependSzToMultiSz(
27813|     IN LPTSTR SzToPrepend,
27814|     IN OUT LPTSTR *MultiSz
27815| );
27816|
27817| STATIC BOOLEAN
27818| AppendSzToMultiSz(
27819|     IN LPTSTR SzToAppend,
27820|     IN OUT LPTSTR *MultiSz
27821| );
27822|
27823| STATIC size_t
27824| MultiSzLength(
27825|     IN LPTSTR MultiSz
27826| );
27827|
27828| STATIC size_t
27829| MultiSzSearchAndDeleteCaseInsensitive(
27830|     IN LPTSTR FindThis,
27831|     IN LPTSTR FindWithin,
27832|     OUT size_t *NewStringLength
27833| );
27834|
27835|
27836| #if DO_NT5_SETUP
27837| /*
27838|  * A wrapper around SetupDiGetDeviceRegistryProperty,
27839|  * | so that I don't have to
27840|  * | deal with memory allocation anywhere else
27841|  * | parameters:

```

```

27842| * DeviceInfoSet - The device information set which
    | contains DeviceInfoData
27843| * DeviceInfoData - Information needed to deal with
    | the given device
27844| * Property      - which property to get (SPDRP_XXX)
27845| * PropertyRegDataType - the type of registry
    | property
27846| */
27847| STATIC PBYTE GetDeviceRegistryProperty(
27848|     IN HDEVINFO DeviceInfoSet,
27849|     IN PSP_DEVINFO_DATA DeviceInfoData,
27850|     IN DWORD Property,
27851|     OUT PDWORD PropertyRegDataType
27852| )
27853| {
27854|     DWORD length = 0;
27855|     PBYTE buffer = NULL;
27856|
27857|     // get the required length of the buffer
27858|     if( W2k->SetupDiGetDeviceRegistryProperty(
        | DeviceInfoSet,
27859|
        | DeviceInfoData,
27860|
        | Property,
27861|
        | NULL, //
        | registry data type
27862|
        | NULL, //
        | buffer
27863|
        | 0, //
        | buffer size
27864|
        | &length //
        | required size
27865|     ) )
27866|     {
27867|         // we should not be successful at this point,
        | so this call succeeding
27868|         // is an error condition
27869|         DLOG(("in GetDeviceRegistryProperty(): "
27870|             "call SetupDiGetDeviceRegistryProperty
        | did not fail\n",
27871|             GetLastError()));
27872|         return (NULL);
27873|     }
27874|
27875|     if( GetLastError() != ERROR_INSUFFICIENT_BUFFER )
27876|     {
27877|         // this means there are no upper filter drivers
        | loaded, so we can just
27878|         // return.
27879|         return (NULL);

```

```

27880| }
27881|
27882| // since we don't have a buffer yet, it is
    | "insufficient"; we allocate
27883| // one and try again.
27884| buffer = malloc( length );
27885| if( buffer == NULL )
27886| {
27887|     DLOG(("in GetDeviceRegistryProperty(): "
27888|         "unable to allocate memory!\n"));
27889|     return (NULL);
27890| }
27891| if( !W2k->SetupDiGetDeviceRegistryProperty(
    | DeviceInfoSet,
27892|
    | DeviceInfoData,
27893|         Property,
27894|
    | PropertyRegDataType,
27895|         buffer,
27896|         length,
27897|         NULL //
    | required size
27898|     ) )
27899| {
27900|     DLOG(("in GetDeviceRegistryProperty(): "
27901|         "couldn't get registry property! error:
    | %i\n",
27902|         GetLastError()));
27903|     free( buffer );
27904|     return (NULL);
27905| }
27906|
27907| // ok, we are finally done, and can return the
    | buffer
27908| return (buffer);
27909| }
27910|
27911|
27912| /*
27913| * restarts the given device
27914| *
27915| * call CM_Query_And_Remove_Subtree (to unload the
    | driver)
27916| * call CM_Reenumerate_DevNode on the _parent_ (to
    | reload the driver)
27917| *
27918| * parameters:
27919| * DeviceInfoSet - The device information set which
    | contains DeviceInfoData

```

```

27920| * DeviceInfoData - Information needed to deal with
    | the given device
27921| */
27922| STATIC BOOLEAN RestartDevice(
27923|     IN HDEVINFO DeviceInfoSet,
27924|     IN OUT PSP_DEVINFO_DATA DeviceInfoData
27925| )
27926| {
27927|     SP_PROPCHANGE_PARAMS params;
27928|     SP_DEVINSTALL_PARAMS installParams;
27929|
27930|     // for future compatibility; this will zero out the
    | entire struct, rather
27931|     // than just the fields which exist now
27932|     memset(&params, 0, sizeof(SP_PROPCHANGE_PARAMS));
27933|
27934|     // initialize the SP_CLASSINSTALL_HEADER struct at
    | the beginning of the
27935|     // SP_PROPCHANGE_PARAMS struct, so that
    | SetupDiSetClassInstallParams will
27936|     // work
27937|     params.ClassInstallHeader.cbSize =
    | sizeof(SP_CLASSINSTALL_HEADER);
27938|     params.ClassInstallHeader.InstallFunction =
    | DIF_PROPERTYCHANGE;
27939|
27940|     // initialize SP_PROPCHANGE_PARAMS such that the
    | device will be stopped.
27941|     params.StateChange = DICS_STOP;
27942|     params.Scope = DICS_FLAG_CONFIGSPECIFIC;
27943|     params.HwProfile = 0; // current profile
27944|
27945|     // prepare for the call to
    | SetupDiCallClassInstaller (to stop the device)
27946|     if( !W2k->SetupDiSetClassInstallParams(
    | DeviceInfoSet,
27947|                                     DeviceInfoData,
27948|     | (PSP_CLASSINSTALL_HEADER) &params,
27949|     | sizeof(SP_PROPCHANGE_PARAMS)
27950|     ) )
27951|     {
27952|         DLOG(("in RestartDevice(): couldn't set the
    | install parameters!"));
27953|         DLOG((" error: %u\n", GetLastError()));
27954|         return (FALSE);
27955|     }
27956|
27957|     // stop the device

```



```

27958|  if( !W2k->SetupDiCallClassInstaller(
      | DIF_PROPERTYCHANGE,
27959|          DeviceInfoSet,
27960|          DeviceInfoData )
27961|  )
27962|  {
27963|      DLOG(("in RestartDevice(): call to class
      | installer (STOP) failed!"));
27964|      DLOG((" error: %u\n", GetLastError() ));
27965|      return (FALSE);
27966|  }
27967|
27968|  // restarting the device
27969|  params.StateChange = DICS_START;
27970|
27971|  // prepare for the call to
      | SetupDiCallClassInstaller (to stop the device)
27972|  if( !W2k->SetupDiSetClassInstallParams(
      | DeviceInfoSet,
27973|          DeviceInfoData,
27974|          (PSP_CLASSINSTALL_HEADER) &params,
27975|          sizeof(SP_PROPCHANGE_PARAMS)
27976|      ) )
27977|  {
27978|      DLOG(("in RestartDevice(): couldn't set the
      | install parameters!"));
27979|      DLOG((" error: %u\n", GetLastError()));
27980|      return (FALSE);
27981|  }
27982|
27983|  // restart the device
27984|  if( !W2k->SetupDiCallClassInstaller(
      | DIF_PROPERTYCHANGE,
27985|          DeviceInfoSet,
27986|          DeviceInfoData )
27987|  )
27988|  {
27989|      DLOG(("in RestartDevice(): call to class
      | installer (START) failed!"));
27990|      DLOG((" error: %u\n", GetLastError()));
27991|      return (FALSE);
27992|  }
27993|
27994|  installParams.cbSize =
      | sizeof(SP_DEVINSTALL_PARAMS);
27995|
27996|  // same as above, the call will succeed, but we
      | still need to check status

```

```

27997|  if( !W2k->SetupDiGetDeviceInstallParams(
      | DeviceInfoSet,
27998|                                DeviceInfoData,
27999|                                &installParams
      | )
28000|  )
28001|  {
28002|      DLOG(("in RestartDevice(): couldn't get the
      | device install params!"));
28003|      DLOG((" error: %u\n", GetLastError() ));
28004|      return (FALSE);
28005|  }
28006|
28007|  // to see if the machine needs to be rebooted
28008|  if( installParams.Flags & DI_NEEDREBOOT )
28009|  {
28010|      return (FALSE);
28011|  }
28012|
28013|  // if we get this far, then the device has been
      | stopped and restarted
28014|  return (TRUE);
28015| }
28016| #endif
28017|
28018| /*
28019|  * prepend the given string to a MultiSz
28020|  *
28021|  * returns true if successful, false if not (will only
      | fail in memory
28022|  * allocation)
28023|  *
28024|  * note: This WILL allocate and free memory, so don't
      | keep pointers to the
28025|  * MultiSz passed in.
28026|  *
28027|  * parameters:
28028|  *  SzToPrepend - string to prepend
28029|  *  MultiSz     - pointer to a MultiSz which will be
      | prepended-to
28030|  */
28031|  STATIC BOOLEAN PrependSzToMultiSz(
28032|      IN  LPTSTR SzToPrepend,
28033|      IN OUT LPTSTR *MultiSz
28034|  )
28035|  {
28036|      size_t szLen;
28037|      size_t multiSzLen;
28038|      LPTSTR newMultiSz = NULL;
28039|

```

```

28040| // get the size, in bytes, of the two buffers
28041| szLen = (_tcslen(SzToPrepend)+1)*sizeof(_TCHAR);
28042| multiSzLen =
    | MultiSzLength(*MultiSz)*sizeof(_TCHAR);
28043| newMultiSz = (LPTSTR)malloc( szLen+multiSzLen );
28044|
28045| if( newMultiSz == NULL ) {
28046|     return (FALSE);
28047| }
28048|
28049| // recopy the old MultiSz into proper position into
    | the new buffer.
28050| // the (char*) cast is necessary, because
    | newMultiSz may be a wchar*, and
28051| // szLen is in bytes.
28052|
28053| memcpy( ((char*)newMultiSz) + szLen, *MultiSz,
    | multiSzLen );
28054|
28055| // copy in the new string
28056| _tcscpy( newMultiSz, SzToPrepend );
28057|
28058| free( *MultiSz );
28059| *MultiSz = newMultiSz;
28060|
28061| return (TRUE);
28062| }
28063|
28064|
28065| /*
28066| * append the given string to a MultiSz
28067| *
28068| * returns true if successful, false if not (will only
    | fail in memory
28069| * allocation)
28070| *
28071| * note: This WILL allocate and free memory, so don't
    | keep pointers to the
28072| * MultiSz passed in.
28073| *
28074| * parameters:
28075| * SzToAppend - string to append
28076| * MultiSz - pointer to a MultiSz which will be
    | appended-to
28077| */
28078| STATIC BOOLEAN AppendSzToMultiSz(
28079|     IN LPTSTR SzToAppend,
28080|     IN OUT LPTSTR *MultiSz
28081| )
28082| {

```

```

28083|  size_t szLen;
28084|  size_t multiSzLen;
28085|  LPTSTR newMultiSz = NULL;
28086|
28087|  // get the size, in bytes, of the two buffers
28088|  szLen = (_tcslen(SzToAppend)+1)*sizeof(_TCHAR);
28089|  multiSzLen =
    | MultiSzLength(*MultiSz)*sizeof(_TCHAR);
28090|  newMultiSz = (LPTSTR)malloc( szLen+multiSzLen );
28091|
28092|  if( newMultiSz == NULL ) {
28093|      return (FALSE);
28094|  }
28095|
28096|  // recopy the old MultiSz into proper position into
    | the new buffer.
28097|
28098|  memcpy( newMultiSz, *MultiSz, multiSzLen );
28099|
28100|  // copy in the new string
28101|  _tcscpy(
    | (_TCHAR*)((char*)newMultiSz)+multiSzLen-sizeof(_TCHAR))
    | , SzToAppend );
28102|
28103|
    | newMultiSz[((szLen+multiSzLen)/sizeof(_TCHAR))-1]=0;
28104|
28105|  free( *MultiSz );
28106|  *MultiSz = newMultiSz;
28107|
28108|  return (TRUE);
28109| }
28110|
28111|
28112| /*
28113|  * returns the length (in characters) of the buffer
    | required to hold this
28114|  * MultiSz, INCLUDING the trailing null.
28115|  *
28116|  * example: MultiSzLength("foo\0bar\0") returns 9
28117|  *
28118|  * note: since MultiSz cannot be null, a number >= 1
    | will always be returned
28119|  *
28120|  * parameters:
28121|  *  MultiSz - the MultiSz to get the length of
28122|  */
28123| STATIC size_t MultiSzLength(
28124|  IN LPTSTR MultiSz
28125|  )

```

```

28126| {
28127|     size_t len = 0;
28128|     size_t totalLen = 0;
28129|
28130|     // search for trailing null character
28131|     while( *MultiSz != _T('\0') ) {
28132|         len = _tcslen(MultiSz)+1;
28133|         MultiSz += len;
28134|         totalLen += len;
28135|     }
28136|
28137|     // add one for the trailing null character
28138|     return (totalLen+1);
28139| }
28140|
28141|
28142| /*
28143|  * Deletes all instances of a string from within a
28144|  * | multi-sz.
28145|  * parameters:
28146|  * FindThis      - the string to find and remove
28147|  * FindWithin    - the string having the instances
28148|  * | removed
28149|  * NewStringLength - the new string length
28150|  */
28151| STATIC size_t MultiSzSearchAndDeleteCaseInsensitive(
28152|     IN LPTSTR FindThis,
28153|     IN LPTSTR FindWithin,
28154|     OUT size_t *NewLength
28155| ) {
28156|     LPTSTR search;
28157|     size_t currentOffset;
28158|     DWORD instancesDeleted;
28159|     size_t searchLen;
28160|
28161|     currentOffset = 0;
28162|     instancesDeleted = 0;
28163|     search = FindWithin;
28164|
28165|     *NewLength = MultiSzLength(FindWithin);
28166|
28167|     // loop while the multisz null terminator is not
28168|     | found
28169|     while ( *search != _T('\0') ) {
28170|         // length of string + null char; used in more
28171|         | than a couple places
28172|         searchLen = _tcslen(search) + 1;

```

```

28172|         // if this string matches the current one in
| the multisz...
28173|         if( _tcsicmp(search, FindThis) == 0 ) {
28174|             // they match, shift the contents of the
| multisz, to overwrite the
28175|             // string (and terminating null), and
| update the length
28176|             instancesDeleted++;
28177|             *NewLength -= searchLen;
28178|             memmove( search, search + searchLen,
| (*NewLength - currentOffset) * sizeof(TCHAR) );
28179|         } else {
28180|             // they don't match, so move pointers,
| increment counters
28181|             currentOffset += searchLen;
28182|             search      += searchLen;
28183|         }
28184|     }
28185|
28186|     return (instancesDeleted);
28187| }
28188|
28189| STATIC ULONG AddUpperFilterToClass( LPTSTR Class,
| LPTSTR UpperFilter )
28190| {
28191|     ULONG Err=ERROR_OUTOFMEMORY;
28192|     HKEY Key;
28193|     TCHAR *UpperFilters;
28194|     DWORD DataSize;
28195|
28196|     UpperFilters = malloc(4096);
28197|     if(UpperFilters) {
28198|
|         _stprintf(UpperFilters,TEXT("SYSTEM\\CurrentControlSet\\
| Control\\Class\\%s"),Class);
28199|         Err = RegOpenKeyEx(
28200|             HKEY_LOCAL_MACHINE, // handle of open key
28201|             UpperFilters, // address of name of
| subkey to open
28202|             0, // dwOptions
28203|             KEY_ALL_ACCESS, // security access mask
28204|             &Key // address of handle of open key
28205|         );
28206|         if(Err==0) {
28207|             memset(UpperFilters,0,4096);
28208|             DataSize = 4096;
28209|             Err = RegQueryValueEx(
28210|                 Key, // handle of key to query
28211|                 TEXT("UpperFilters"), // address of
| name of value to query

```

```

28212|         NULL, // reserved
28213|         NULL, // address of buffer for value
      | type
28214|         (char*)UpperFilters, // address of
      | data buffer
28215|         &DataSize // address of data buffer
      | size
28216|     );
28217|     if(Err!=5) {
28218|         DLOG(("Key read with %08x
      | error\n",Err));
28219|         // okay add driver
28220|
      | if(PrependSzToMultiSz(UpperFilter,&UpperFilters)) {
28221|         Err =
      | RegSetValueEx(Key,TEXT("UpperFilters"),0,REG_MULTI_SZ,(B
      | YTE*)UpperFilters,MultiSzLength(UpperFilters));
28222|     } else {
28223|         Err = GetLastError();
28224|     }
28225| } else {
28226|     DLOG(("Error Access Denied accessing
      | key\n"));
28227| }
28228| // close the registry
28229| RegCloseKey(Key);
28230| } else {
28231|     DLOG(("Error %08x opening key\n",Err));
28232| }
28233| free(UpperFilters);
28234| } else {
28235|     DLOG(("Error Out of memory\n"));
28236|     Err = ERROR_OUTOFMEMORY;
28237| }
28238| return Err;
28239| }
28240|
28241| STATIC ULONG RemoveUpperFilterFromClass( LPTSTR Class,
      | LPTSTR UpperFilter )
28242| {
28243|     ULONG Err=ERROR_OUTOFMEMORY;
28244|     HKEY Key;
28245|     TCHAR *UpperFilters;
28246|     DWORD DataSize;
28247|
28248|     UpperFilters = malloc(4096);
28249|     if(UpperFilters) {
28250|         | _sprintf(UpperFilters,TEXT("SYSTEM\\CurrentControlSet\\
      | Control\\Class\\%s"),Class);

```

```

28251|     Err = RegOpenKeyEx(
28252|         HKEY_LOCAL_MACHINE, // handle of open key
28253|         UpperFilters, // address of name of
        | subkey to open
28254|         0, // dwOptions
28255|         KEY_ALL_ACCESS, // security access mask
28256|         &Key // address of handle of open key
28257|     );
28258|     if(Err==0) {
28259|         memset(UpperFilters,0,4096);
28260|         DataSize = 4096;
28261|         Err = RegQueryValueEx(
28262|             Key, // handle of key to query
28263|             TEXT("UpperFilters"), // address of
        | name of value to query
28264|             NULL, // reserved
28265|             NULL, // address of buffer for value
        | type
28266|             (char*)UpperFilters, // address of
        | data buffer
28267|             &DataSize // address of data buffer
        | size
28268|         );
28269|         if(Err!=5) {
28270|             DLOG(("Key read with %08x
        | error\n",Err));
28271|
28272|             MultiSzSearchAndDeleteCaseInsensitive(
        | UpperFilter, UpperFilters, &DataSize);
28273|             Err =
        | RegSetValueEx(Key,TEXT("UpperFilters"),0,REG_MULTI_SZ,(B
        |YTE*)UpperFilters,DataSize);
28274|         } else {
28275|             DLOG(("Error Access Denied accessing
        | key\n"));
28276|         }
28277|         // close the registry
28278|         RegCloseKey(Key);
28279|     } else {
28280|         DLOG(("Error %08x opening key\n",Err));
28281|     }
28282|     free(UpperFilters);
28283| } else {
28284|     DLOG(("Error Out of memory\n"));
28285|     Err = ERROR_OUTOFMEMORY;
28286| }
28287| return Err;
28288| }
28289|
28290| STATIC ULONG AddExeToBootExecute( LPTSTR Exe )

```



```

28291| {
28292|     ULONG Err=ERROR_OUTOFMEMORY;
28293|     HKEY Key;
28294|     TCHAR *BootExecute;
28295|     DWORD DataSize;
28296|
28297|     BootExecute= malloc(4096);
28298|     if(BootExecute) {
28299|
28300|         _sprintf(BootExecute,TEXT("SYSTEM\\CurrentControlSet\\C
28301|         ontrol\\Session Manager"));
28302|         Err = RegOpenKeyEx(
28303|             HKEY_LOCAL_MACHINE, // handle of open key
28304|             BootExecute, // address of name of
28305|             | subkey to open
28306|             0, // dwOptions
28307|             KEY_ALL_ACCESS, // security access mask
28308|             &Key // address of handle of open key
28309|         );
28310|         if(Err==0) {
28311|             memset(BootExecute,0,4096);
28312|             DataSize = 4096;
28313|             Err = RegQueryValueEx(
28314|                 Key, // handle of key to query
28315|                 TEXT("BootExecute"), // address of
28316|                 | name of value to query
28317|                 NULL, // reserved
28318|                 NULL, // address of buffer for value
28319|                 | type
28320|                 (char*)BootExecute, // address of data
28321|                 | buffer
28322|                 &DataSize // address of data buffer
28323|                 | size
28324|             );
28325|             if(Err!=5) {
28326|                 DLOG(("Key read with %08x
28327|                 | error\n",Err));
28328|                 // okay add driver
28329|                 if(AppendSzToMultiSz(Exe,&BootExecute))
28330|                 | {
28331|                     Err =
28332|                     | RegSetValueEx(Key,TEXT("BootExecute"),0,REG_MULTI_SZ,(BY
28333|                     | TE*)BootExecute,MultiSzLength(BootExecute));
28334|                 } else {
28335|                     Err = GetLastError();
28336|                 }
28337|             } else {
28338|                 DLOG(("Error Access Denied accessing
28339|                 | key\n"));
28340|             }

```

```

28329|         // close the registry
28330|         RegCloseKey(Key);
28331|     } else {
28332|         DLOG(("Error %08x opening key\n",Err));
28333|     }
28334|     free(BootExecute);
28335| } else {
28336|     DLOG(("Error Out of memory\n"));
28337|     Err = ERROR_OUTOFMEMORY;
28338| }
28339| return Err;
28340| }
28341|
28342| STATIC ULONG RemoveExeToBootExecute( LPTSTR Exe )
28343| {
28344|     ULONG Err=ERROR_OUTOFMEMORY;
28345|     HKEY Key;
28346|     TCHAR *BootExecute;
28347|     DWORD DataSize;
28348|
28349|     BootExecute= malloc(4096);
28350|     if(BootExecute) {
28351|
28352|         | _sprintf(BootExecute,TEXT("SYSTEM\\CurrentControlSet\\C
28353|         | ontrol\\Session Manager"));
28354|         Err = RegOpenKeyEx(
28355|             HKEY_LOCAL_MACHINE, // handle of open key
28356|             BootExecute, // address of name of
28357|             | subkey to open
28358|             0, // dwOptions
28359|             KEY_ALL_ACCESS, // security access mask
28360|             &Key // address of handle of open key
28361|         );
28362|         if(Err==0) {
28363|             memset(BootExecute,0,4096);
28364|             DataSize = 4096;
28365|             Err = RegQueryValueEx(
28366|                 Key, // handle of key to query
28367|                 TEXT("BootExecute"), // address of
28368|                 | name of value to query
28369|                 NULL, // reserved
28370|                 NULL, // address of buffer for value
28371|                 | type
28372|                 (char*)BootExecute, // address of data
28373|                 | buffer
28374|                 &DataSize // address of data buffer
28375|                 | size
28376|             );
28377|             if(Err!=5) {
28378|                 DLOG(("Key read with %08x

```

```

    | error\n",Err));
28372|
28373|         MultiSzSearchAndDeleteCaseInsensitive(
    | Exe, BootExecute, &DataSize);
28374|         Err =
    | RegSetValueEx(Key,TEXT("BootExecute"),0,REG_MULTI_SZ,(BY
    | TE*)BootExecute,DataSize);
28375|     } else {
28376|         DLOG(("Error Access Denied accessing
    | key\n"));
28377|     }
28378|     // close the registry
28379|     RegCloseKey(Key);
28380| } else {
28381|     DLOG(("Error %08x opening key\n",Err));
28382| }
28383| free(BootExecute);
28384| } else {
28385|     DLOG(("Error Out of memory\n"));
28386|     Err = ERROR_OUTOFMEMORY;
28387| }
28388| return Err;
28389| }
28390|
28391|
28392|
28393| #if DO_NT5_SETUP
28394| STATIC void ImportW2kImports( )
28395| {
28396|     W2k = malloc(sizeof(*W2k));
28397|     if(W2k) {
28398|         SetupDll = LoadLibrary(TEXT("setupapi.dll"));
28399|         if((int)SetupDll>31) {
28400|             W2k->SetupDiDestroyDeviceInfoList =
    | (void*)GetProcAddress(SetupDll,"SetupDiDestroyDeviceInfo
    | List");
28401|             W2k->SetupDiEnumDeviceInfo =
    | (void*)GetProcAddress(SetupDll,"SetupDiEnumDeviceInfo");
28402|             W2k->SetupDiCallClassInstaller =
    | (void*)GetProcAddress(SetupDll,"SetupDiCallClassInstalle
    | r");
28403|
28404| #ifdef UNICODE
28405|             W2k->SetupDiGetClassDevs =
    | (void*)GetProcAddress(SetupDll,"SetupDiGetClassDevsW");
28406|             W2k->SetupDiGetDeviceInstallParams =
    | (void*)GetProcAddress(SetupDll,"SetupDiGetDeviceInstallP
    | aramsW");
28407|             W2k->SetupDiSetClassInstallParams =
    | (void*)GetProcAddress(SetupDll,"SetupDiSetClassInstallPa

```

```

    | ramsW");
28408|     W2k->SetupDiGetDeviceRegistryProperty  =
    | (void*)GetProcAddress(SetupDll,"SetupDiGetDeviceRegistry
    | PropertyW");
28409| #else
28410|     W2k->SetupDiGetClassDevs                =
    | (void*)GetProcAddress(SetupDll,"SetupDiGetClassDevsA");
28411|     W2k->SetupDiGetDeviceInstallParams      =
    | (void*)GetProcAddress(SetupDll,"SetupDiGetDeviceInstallP
    | aramsA");
28412|     W2k->SetupDiSetClassInstallParams        =
    | (void*)GetProcAddress(SetupDll,"SetupDiSetClassInstallPa
    | ramsA");
28413|     W2k->SetupDiGetDeviceRegistryProperty  =
    | (void*)GetProcAddress(SetupDll,"SetupDiGetDeviceRegistry
    | PropertyA");
28414| #endif
28415|     }
28416| }
28417| }
28418|
28419| STATIC void UnimportW2kImports()
28420| {
28421|     if(W2k) {
28422|         FreeLibrary(SetupDll);
28423|         free(W2k);
28424|         W2k = NULL;
28425|         SetupDll = INVALID_HANDLE_VALUE;
28426|     }
28427| }
28428| #endif
28429|
28430| /*
28431| Will do what is needed to get PSM installed on
    | Win2k.
28432| */
28433|
28434| ULONG SetUpForWin2k( BOOLEAN *RebootNeeded )
28435| {
28436| #if DO_NT5_SETUP
28437|     // these two constants are used to help enumerate
    | through the list of all
28438|     // disks and volumes on the system. Adding another
    | GUID should "just work"
28439|     static const GUID * deviceGuids[] = {
28440| //         &GUID_DEVCLASS_VOLUME,
28441|         &GUID_DEVCLASS_DISKDRIVE
28442|     };
28443|     static const int numdeviceGuids =
    | sizeof(deviceGuids) / sizeof(LPGUID);

```

```

28444|  HDEVINFO          devInfo =
      | INVALID_HANDLE_VALUE;
28445|  SP_DEVINFO_DATA    devInfoData;
28446|  int devGuidIndex;
28447|  int deviceIndex;
28448| #endif
28449|  ULONG Err=0;
28450|
28451|  *RebootNeeded = FALSE;
28452|
28453|  DLOG(("Installing service\n"));
28454|  // install device into registry
28455|  Err = Svc_InstallService(
28456|      TEXT("System32\\DRIVERS\\psman5.sys"),
28457|      TEXT("psman5"),
28458|      TEXT("Persistent Storage Manager"),
28459|      NULL,
28460|      TEXT("Filter"),
28461|      1,
28462|      0,
28463|      5
28464|  );
28465|
28466|  if(Err!=0) {
28467|      DLOG(("Error %08x installing service\n",Err));
28468|      return Err;
28469|  }
28470|
28471|  DLOG(("Adding UpperFilter to Volume class\n"));
28472|  // make registry changes now.
28473|  // Volume Class
28474|  Err =
      | AddUpperFilterToClass(TEXT("{71a27cdd-812a-11d0-bec7-080
      | 02be2092f}"),TEXT("psman5"));
28475|
28476|  DLOG(("Adding UpperFilter to Disk class\n"));
28477|  // DiskDrive class
28478|  Err =
      | AddUpperFilterToClass(TEXT("{4d36e967-e325-11ce-bfc1-080
      | 02be10318}"),TEXT("psman5"));
28479|
28480|  DLOG(("Adding exe to boot execute\n"));
28481|  Err = AddExeToBootExecute(TEXT("psmready"));
28482|
28483|  DLOG(("Adding Log source\n"));
28484|  // Add event log source
28485|  Err = AddLogSource(TEXT("psman5"));
28486|
28487|  // always reboot now.
28488|  *RebootNeeded = TRUE;

```

```

28489|
28490| #if DO_NT5_SETUP
28491|     DLOG(("Starting driver\n"));
28492|     Err = Svc_StartService(TEXT("psman5"));
28493|
28494|     if(Err) {
28495|         DLOG(("Error %08x starting service\n",Err));
28496|         *RebootNeeded = TRUE;
28497|         //return Err;
28498|     }
28499|     Err = 0;
28500|     ImportW2kImports();
28501|
28502|     // okay now we need to loop through the devices and
        | restart them
28503|     // as the filter chain has changed.
28504|
28505|     DLOG(("Restarting devices\n"));
28506|     // This outer loop steps through the array of
        | device guid pointers that is
28507|     // defined above main(). It was just the easiest
        | way to deal with both
28508|     // Disks and Volumes (and it is easy to add other
        | types of devices)
28509|     for(devGuidIndex = 0; devGuidIndex<numdeviceGuids;
        | devGuidIndex++) {
28510|         // get a list of devices which support the
        | given interface
28511|         devInfo = W2k->SetupDiGetClassDevs(
        | deviceGuids[devGuidIndex],
28512|             NULL,
28513|             NULL,
28514|             DIGCF_PROFILE |
28515|             //
        | DIGCF_DEVICEINTERFACE |
28516|             DIGCF_PRESENT );
28517|
28518|         if( devInfo == INVALID_HANDLE_VALUE ) {
28519|             Err = GetLastError();
28520|             DLOG(("Setup: Error = %08x!\n",Err));
28521|             *RebootNeeded = TRUE;
28522|             break;
28523|         }
28524|
28525|         // as per DDK docs on SetupDiEnumDeviceInfo
28526|         devInfoData.cbSize = sizeof(SP_DEVINFO_DATA);
28527|
28528|         // step through the list of devices for this
        | handle
28529|         // get device info at index deviceIndex, the

```

```

| function returns FALSE
28530|     // when there is no device at the given index.
28531|     for( deviceIndex=0;
28532|         W2k->SetupDiEnumDeviceInfo( devInfo,
| deviceIndex, &devInfoData );
28533|         deviceIndex++
28534|     ) {
28535|         if( !RestartDevice( devInfo, &devInfoData)
| ) {
28536|             *RebootNeeded = TRUE;
28537|         }
28538|     }
28539|
28540|     // clean up the device list
28541|     if( devInfo != INVALID_HANDLE_VALUE ) {
28542|         if( !W2k->SetupDiDestroyDeviceInfoList(
| devInfo ) ) {
28543|             Err = GetLastError();
28544|             DLOG(("unable to delete device info
| list! error: %u\n",Err));
28545|         }
28546|     }
28547| }
28548|
28549| UnimportW2kImports();
28550| // we should be set up at this point whether a
| reboot is needed or not
28551| #endif
28552| return 0;
28553| }
28554|
28555| ULONG UnSetUpForWin2k( BOOLEAN *RebootNeeded )
28556| {
28557| #if DO_NT5_SETUP
28558|     // these two constants are used to help enumerate
| through the list of all
28559|     // disks and volumes on the system. Adding another
| GUID should "just work"
28560|     static const GUID * deviceGuids[] = {
28561| //         &GUID_DEVCLASS_VOLUME,
28562|         &GUID_DEVCLASS_DISKDRIVE
28563|     };
28564|     static const int numdeviceGuids =
| sizeof(deviceGuids) / sizeof(LPGUID);
28565|     HDEVINFO         devInfo =
| INVALID_HANDLE_VALUE;
28566|     SP_DEVINFO_DATA   devInfoData;
28567|     int devGuidIndex;
28568|     int deviceIndex;
28569| #endif

```

```

28570|  ULONG Err=0;
28571|
28572|  *RebootNeeded = FALSE;
28573|
28574|
    | RemoveUpperFilterFromClass(TEXT("{71a27cdd-812a-11d0-bec
    | 7-08002be2092f}"),TEXT("psman5"));
28575|
    | RemoveUpperFilterFromClass(TEXT("{4d36e967-e325-11ce-bfc
    | 1-08002be10318}"),TEXT("psman5"));
28576|  RemoveLogSource(TEXT("psman5"));
28577|  RemoveExeToBootExecute(TEXT("psmready"));
28578|
28579|  // we do not current support stopping, but it we do
    | in the
28580|  // future, lets support it.
28581|  #if DO_NT5_SETUP
28582|  Err = Svc_StopService(TEXT("psman5"));
28583|  #endif
28584|  Err = Svc_RemoveService(TEXT("psman5"));
28585|
28586|  *RebootNeeded = TRUE;
28587|
28588|  Err = 0;
28589|
28590|  #if DO_NT5_SETUP
28591|  ImportW2kImports();
28592|
28593|  // okay now we need to loop through the devices and
    | restart them
28594|  // as the filter chain has changed.
28595|
28596|  // This outer loop steps through the array of
    | device guid pointers that is
28597|  // defined above main(). It was just the easiest
    | way to deal with both
28598|  // Disks and Volumes (and it is easy to add other
    | types of devices)
28599|  for(devGuidIndex = 0; devGuidIndex<numdeviceGuids;
    | devGuidIndex++) {
28600|      // get a list of devices which support the
    | given interface
28601|      devInfo = W2k->SetupDiGetClassDevs(
    | deviceGuids[devGuidIndex],
28602|      NULL,
28603|      NULL,
28604|      DIGCF_PROFILE |
28605|  //
    | DIGCF_DEVICEINTERFACE |
28606|      DIGCF_PRESENT );

```



```

28607|
28608|     if( devInfo == INVALID_HANDLE_VALUE ) {
28609|         Err = GetLastError();
28610|         DLOG(("Setup: Error = %08x!\n",Err));
28611|         *RebootNeeded = TRUE;
28612|         break;
28613|     }
28614|
28615|     // as per DDK docs on SetupDiEnumDeviceInfo
28616|     devInfoData.cbSize = sizeof(SP_DEVINFO_DATA);
28617|
28618|     // step through the list of devices for this
    | handle
28619|     // get device info at index deviceIndex, the
    | function returns FALSE
28620|     // when there is no device at the given index.
28621|     for( deviceIndex=0;
28622|         W2k->SetupDiEnumDeviceInfo( devInfo,
    | deviceIndex, &devInfoData );
28623|         deviceIndex++
28624|     ) {
28625|         if( !RestartDevice( devInfo, &devInfoData)
    | ) {
28626|             *RebootNeeded = TRUE;
28627|         }
28628|     }
28629|
28630|     // clean up the device list
28631|     if( devInfo != INVALID_HANDLE_VALUE ) {
28632|         if( !W2k->SetupDiDestroyDeviceInfoList(
    | devInfo ) ) {
28633|             Err = GetLastError();
28634|             DLOG(("unable to delete device info
    | list! error: %u\n",Err));
28635|         }
28636|     }
28637| }
28638|
28639| UnimportW2kImports();
28640| #endif
28641| return 0;
28642| }
28643|
28644|
28645|
28646| File Listing: SETUP5.h
28647|
28648| ULONG SetUpForWin2k( BOOLEAN *RebootNeeded );
28649| ULONG UnSetUpForWin2k( BOOLEAN *RebootNeeded );
28650|

```

```

28651|
28652|
28653| File Listing: senum.c
28654|
28655| #include <stdio.h>
28656| #include <stdlib.h>
28657| #include <string.h>
28658| #include <stddef.h>
28659| #include <windows.h>
28660| #include <tchar.h>
28661| #include <process.h>
28662| #include <time.h>
28663| #include <direct.h>
28664| #include <lm.h>
28665|
28666| #include <winioctl.h>
28667| #include <undoc.h>
28668| // psm api
28669| #include <psm.h>
28670| // ioctls we need to send down
28671| #include "..\driver\ioctl.h"
28672|
28673| #include "volume.h"
28674|
28675| #include "defrag.h"
28676| #include <mountmgr.h>
28677| #include <ntddstor.h>
28678| #include <ntddvol.h>
28679|
28680| #include "setup5.h"
28681| #include "setup4.h"
28682| #include "service.h"
28683|
28684|
28685| //-----
| -----
28686|
28687|
28688| PSMSTATUS PSMAPI Psm_GetNumberOfActiveSnapshots (
28689|     OUT ULONG *numberOfSnapshots )
28690| {
28691|     PSMSTATUS err = 0;
28692|     *numberOfSnapshots = 0;
28693|     return err;
28694| }
28695|
28696|
28697| //-----
| -----
28698|

```

```

28699|
28700| PSMSTATUS PSMAPI Psm_GetActiveSnapshotPointer (
28701|     IN ULONG snapshotIndex,      // must be in the
        | range 0..(numberOfSnapshots-1)
28702|     OUT PVOID *snapshotPointer )
28703| {
28704|     snapshotPointer = NULL;
28705|     return 0;
28706| }
28707|
28708|
28709| //-----
        | -----
28710|
28711|
28712|
28713| /*--- end of file ssenum.c ---*/
28714|
28715|
28716|
28717| File Listing: trust.c
28718|
28719| #include <stdio.h>
28720| #include <stdlib.h>
28721| #include <string.h>
28722| #include <stddef.h>
28723| #include <windows.h>
28724| #include <tchar.h>
28725| #include <process.h>
28726| #include <time.h>
28727| #include <direct.h>
28728| #include <lm.h>
28729|
28730| #include <winioctl.h>
28731| // psm api
28732| #include <psm.h>
28733| #include <wintrust.h>
28734| #include <ntsecapi.h>
28735| #include <psmoem.h>
28736|
28737| #ifdef _DEBUG
28738| void PSM_LogDebugInfo( const TCHAR *fmt,...);
28739| #define DLOG(x) PSM_LogDebugInfo x
28740| #else
28741| #define DLOG(x)
28742| #endif
28743|
28744| extern DWORD ExceptionFilter( EXCEPTION_POINTERS *ep );
28745|
28746| ULONG GetOurPath( WCHAR *OurPath );

```

```

28747|
28748| typedef HRESULT (WINAPI *WINVERIFYTRUST)(HWND hwnd,
| GUID *ActionID, LPVOID ActionData);
28749|
28750| #define WIN_SPUB_ACTION_PUBLISHED_SOFTWARE_NOBADUI
| {0xc6b2e8d0, 0xe005, 0x11cf, { 0xa1, 0x34, 0x0, 0xc0,
| 0x4f,0xd7, 0xbf, 0x43 } }
28751|
28752| #define PSM_MODULE_NAME    L"psmlapi.dll"
28753| #define PSM_OEM_NAME      L"psmprov.dll"
28754|
28755| // #define NUMBER_OF_DAYS_IN_EVAL  120 -- Moved to
| psmoem.h to be vendor specific
28756|
28757| const WCHAR * const SearchModules[] = {
28758|     PSM_MODULE_NAME,
28759|     PSM_OEM_NAME,
28760|     L"drbackup.dll"
28761| };
28762|
28763| const WCHAR * const DirectModules[] = {
28764|     L"\\psmready.exe",
28765|     L"\\drivers\\psman5.sys"
28766| };
28767|
28768| #define NUM_SEARCH_MODULES
| (sizeof(SearchModules)/sizeof(SearchModules[0]))
28769| #define NUM_DIRECT_MODULES
| (sizeof(DirectModules)/sizeof(DirectModules[0]))
28770|
28771| BOOLEAN IsSoftwareTrusted()
28772| {
28773|     WINVERIFYTRUST pWinVerifyTrust= NULL;
28774|     HINSTANCE WinTrust=
| LoadLibraryW(WT_PROVIDER_DLL_NAME);
28775|     ULONG Status;
28776|     WCHAR FullPath[1024];
28777|     WCHAR FileName[1024];
28778|     ULONG DirectCount=0;
28779|     ULONG SearchCount=0;
28780|     ULONG MissingOem=0;
28781|     ULONG i;
28782|     ULONG Err;
28783|     {
28784|         WCHAR File[1024];
28785|         WCHAR Dir[1024];
28786|         WCHAR Drive[256];
28787|
28788|         Err = GetOurPath(File);
28789|         if(Err) {

```

```

28790|         return FALSE;
28791|     }
28792|
28793|     DLOG(("File: %S\n",File));
28794|
28795|     _wsplitpath( File, Drive, Dir, NULL, NULL);
28796|     wcscpy(FullPath,Drive);
28797|     wcscat(FullPath,Dir);
28798|     DLOG(("Drive %S\n",Drive));
28799|     DLOG(("Dir %S\n",Dir));
28800| }
28801|
28802| if(WinTrust!=INVALID_HANDLE_VALUE) {
28803|     pWinVerifyTrust =
        | (WINVERIFYTRUST)GetProcAddress(WinTrust,"WinVerifyTrust"
        | );
28804|     if(pWinVerifyTrust) {
28805|         GUID PublishedSoftware =
        | WIN_SPUB_ACTION_PUBLISHED_SOFTWARE;
28806|         GUID SubjectPelmage =
        | WIN_TRUST_SUBJECTYPE_PE_IMAGE;
28807|         GUID *ActionGUID=&PublishedSoftware;
28808|         WIN_TRUST_SUBJECT_FILE Subject={0};
28809|         WIN_TRUST_ACTDATA_CONTEXT_WITH_SUBJECT
        | ActionData={0};
28810|
28811|         Subject.lpPath = FileName;
28812|         Subject.hFile = INVALID_HANDLE_VALUE;
28813|         ActionData.Subject = &Subject;
28814|         ActionData.hClientToken = NULL;
28815|         ActionData.SubjectType = &SubjectPelmage;
28816|
28817|         for(i=0;i<NUM_SEARCH_MODULES;i++) {
28818|             WCHAR *p;
28819|             WCHAR Buffer[1024];
28820|             wcscpy(FileName,FullPath);
28821|             wcscat(FileName,SearchModules[i]);
28822|
28823|             // check directory where this dll is
        | first
28824|
        | if(!SearchPathW(NULL,FileName,NULL,1024,Buffer,&p)) {
28825|                 // not found, lets search the path
28826|
        | SearchPathW(NULL,SearchModules[i],NULL,1024,FileName,&p)
        | ;
28827|         }
28828|         Status =
        | pWinVerifyTrust((HWND)INVALID_HANDLE_VALUE, ActionGUID,
        | &ActionData );

```



```

28866|     return FALSE;
28867| }
28868| }
28869|
28870| Redacted. Functions to secure and identify evaluation
    | and fully functioning versions. Not required to teach
    | the invention. --LPW
28871|
28872|
28873|
28874| File Listing: VOLUME.c
28875|
28876| #include <stdio.h>
28877| #include <stdlib.h>
28878| #include <stdarg.h>
28879| #include <string.h>
28880| #include <time.h>
28881| #include <process.h>
28882| #include <io.h>
28883| #include <errno.h>
28884| #include <conio.h>
28885| #include <fcntl.h>
28886| #include <windows.h>
28887| #include <windowsx.h>
28888| #include <commctrl.h> // includes the common control
    | header
28889| #include <tchar.h>
28890| #include <winioctl.h>
28891| #include <ntddscsi.h>
28892| #include <lm.h>
28893| #include <tchar.h>
28894|
28895| #include <undoc.h>
28896| #include <psm.h>
28897| #include "..\driver\ioctl.h"
28898| #include <aclapi.h>
28899| #include "dlog.h"
28900|
28901| ULONG VDisk_UnWriteProtect ( CHAR DriveLetter )
28902| {
28903|     HANDLE hVolume;
28904|     TCHAR szVolumeName[8];
28905|     DWORD dwAccessFlags;
28906|     ULONG Err = 0;
28907|     ULONG dwBytesReturned;
28908|
28909|     dwAccessFlags = 0; //GENERIC_READ;
28910|
28911|     wsprintf(szVolumeName, TEXT("\\\\.\\%c:"),
    | DriveLetter);

```

```

28912| hVolume = CreateFile( szVolumeName,
28913|     dwAccessFlags,
28914|     FILE_SHARE_READ | FILE_SHARE_WRITE,
28915|     NULL,
28916|     OPEN_EXISTING,           0,
28917|     NULL );
28918|
28919| if(hVolume) {
28920|     if (!DeviceIoControl(hVolume,
28921|         IOCTL_UNWRITE_PROTECT,
28922|         NULL, 0,
28923|         NULL, 0,
28924|         &dwBytesReturned,
28925|         NULL)) {
28926|
28927|         Err = GetLastError();
28928|     }
28929|
28930|     CloseHandle(hVolume);
28931| } else {
28932|     Err = GetLastError();
28933| }
28934| return Err;
28935| }
28936|
28937| ULONG VDisk_WriteProtect ( CHAR DriveLetter )
28938| {
28939|     HANDLE hVolume;
28940|     TCHAR szVolumeName[8];
28941|     DWORD dwAccessFlags;
28942|     ULONG dwBytesReturned;
28943|     ULONG Err = 0;
28944|
28945|     dwAccessFlags = 0; //GENERIC_READ;
28946|
28947|     wsprintf(szVolumeName, TEXT("\\\\.\\%c:"),
28948|         | DriveLetter);
28949|     hVolume = CreateFile( szVolumeName,
28950|         dwAccessFlags,
28951|         FILE_SHARE_READ | FILE_SHARE_WRITE,
28952|         NULL,
28953|         OPEN_EXISTING,           0,
28954|         NULL );
28955|
28956| if(hVolume) {
28957|     if (!DeviceIoControl(hVolume,
28958|         IOCTL_WRITE_PROTECT,
28959|         NULL, 0,
28960|         NULL, 0,
28961|         &dwBytesReturned,

```



```

28961|         NULL)) {
28962|
28963|         Err = GetLastError();
28964|     }
28965|
28966|     CloseHandle(hVolume);
28967| } else {
28968|     Err = GetLastError();
28969| }
28970| return Err;
28971| }
28972|
28973| HANDLE OpenVolume(TCHAR cDriveLetter, DWORD
| dwAccessFlags )
28974| {
28975|     HANDLE hVolume;
28976|     TCHAR szVolumeName[8];
28977|     TCHAR szRootName[5];
28978|
28979|     wsprintf(szRootName, TEXT("%c:\\"), cDriveLetter);
28980|
28981|     wsprintf(szVolumeName, TEXT("\\\\.\%c:"),
| cDriveLetter);
28982|     hVolume = CreateFile( szVolumeName,
28983|         dwAccessFlags,
28984|         FILE_SHARE_READ | FILE_SHARE_WRITE,
28985|         NULL,
28986|         OPEN_EXISTING,
28987|         0,
28988|         NULL );
28989|     if(hVolume==INVALID_HANDLE_VALUE) {
28990|         DLOG((TEXT("Error %08x opening volume for
| %08x\n"), GetLastError(),dwAccessFlags));
28991|     }
28992|     return hVolume;
28993| }
28994|
28995| BOOL CloseVolume(HANDLE hVolume)
28996| {
28997|     return CloseHandle(hVolume);
28998| }
28999|
29000| #define LOCK_TIMEOUT    10000    // 10 Seconds
29001| #define LOCK_RETRIES    20
29002|
29003| BOOL LockVolume(HANDLE hVolume)
29004| {
29005|     DWORD dwBytesReturned=0;
29006|     DWORD dwSleepAmount = LOCK_TIMEOUT / LOCK_RETRIES;
29007|     int nTryCount;

```

```

29008|
29009| // Do this in a loop until a timeout period has
      | expired
29010|
29011| for (nTryCount = 0; nTryCount < LOCK_RETRIES;
      | nTryCount++) {
29012|     if (DeviceIoControl(hVolume,
29013|         FSCTL_LOCK_VOLUME,
29014|         NULL, 0,
29015|         NULL, 0,
29016|         &dwBytesReturned,
29017|         NULL))
29018|         return TRUE;
29019|     DLOG((TEXT("Try %d: Error %08x locking
      | volume\n"), nTryCount, GetLastError()));
29020|     Sleep(dwSleepAmount);
29021| }
29022| return FALSE;
29023| }
29024|
29025| BOOL UnlockVolume(HANDLE hVolume)
29026| {
29027|     DWORD dwBytesReturned=0;
29028|     DWORD dwSleepAmount = LOCK_TIMEOUT / LOCK_RETRIES;
29029|     int nTryCount;
29030|
29031|     // Do this in a loop until a timeout period has
      | expired
29032|
29033|     for (nTryCount = 0; nTryCount < LOCK_RETRIES;
      | nTryCount++) {
29034|         if (DeviceIoControl(hVolume,
29035|             FSCTL_UNLOCK_VOLUME,
29036|             NULL, 0,
29037|             NULL, 0,
29038|             &dwBytesReturned,
29039|             NULL))
29040|             return TRUE;
29041|         DLOG((TEXT("Try %d: Error %08x unlocking
      | volume\n"), nTryCount, GetLastError()));
29042|         Sleep(dwSleepAmount);
29043|     }
29044|     return FALSE;
29045| }
29046|
29047| BOOL DismountVolume(HANDLE hVolume)
29048| {
29049|     DWORD dwBytesReturned=0;
29050|     BOOL B;
29051|

```

```

29052| B = DeviceIoControl( hVolume,
29053|     FSCTL_DISMOUNT_VOLUME,
29054|     NULL, 0,
29055|     NULL, 0,
29056|     &dwBytesReturned,
29057|     NULL);
29058|
29059| if(!B) {
29060|     DLOG((TEXT("Error %08x dismounting volume\n"),
29061|         | GetLastError()));
29062| }
29063| return B;
29064| }
29065| BOOL PreventRemovalOfVolume(HANDLE hVolume, BOOL
29066|     | fPreventRemoval)
29067| {
29068|     DWORD dwBytesReturned=0;
29069|     PREVENT_MEDIA_REMOVAL PMRBuffer;
29070|     PMRBuffer.PreventMediaRemoval =
29071|         | (BOOLEAN)fPreventRemoval;
29072|     return DeviceIoControl( hVolume,
29073|         | IOCTL_DISK_MEDIA_REMOVAL, //
29074|         | IOCTL_STORAGE_MEDIA_REMOVAL,
29075|         &PMRBuffer,
29076|         sizeof(PREVENT_MEDIA_REMOVAL),
29077|         NULL, 0,
29078|         &dwBytesReturned,
29079|         NULL);
29080| }
29081|
29082| BOOL AutoEjectVolume(HANDLE hVolume)
29083| {
29084|     DWORD dwBytesReturned;
29085|     return DeviceIoControl( hVolume,
29086|         | IOCTL_DISK_EJECT_MEDIA, //
29087|         | IOCTL_STORAGE_EJECT_MEDIA,
29088|         NULL, 0,
29089|         NULL, 0,
29090|         &dwBytesReturned,
29091|         NULL);
29092| }
29093|
29094| BOOL EjectVolume(TCHAR cDriveLetter)
29095| {
29096|     HANDLE hVolume;
29097|     BOOL fRemoveSafely = FALSE;
29098|     BOOL fAutoEject = FALSE;
29099|

```

```

29097|   hVolume = OpenVolume(cDriveLetter, GENERIC_READ |
|   GENERIC_WRITE);
29098|   if (hVolume == INVALID_HANDLE_VALUE) {
29099|       hVolume = OpenVolume(cDriveLetter, GENERIC_READ
|   );
29100|       if (hVolume == INVALID_HANDLE_VALUE) {
29101|           hVolume = OpenVolume(cDriveLetter, 0);
29102|           if (hVolume == INVALID_HANDLE_VALUE) {
29103|               return FALSE;
29104|           }
29105|       }
29106|   }
29107|
29108|   // Lock and dismount the volume.
29109|   if (LockVolume(hVolume) && DismountVolume(hVolume))
|   {
29110|       fRemoveSafely = TRUE;
29111|       // Set prevent removal to false and eject the
|   volume.
29112|       if (PreventRemovalOfVolume(hVolume, FALSE) &&
|   AutoEjectVolume(hVolume))
29113|           fAutoEject = TRUE;
29114|   }
29115|
29116|   // Close the volume so other processes can use the
|   drive.
29117|   if (!CloseVolume(hVolume))
29118|       return FALSE;
29119|
29120|   // if the media was dismounted ok, return true
29121|   if(fRemoveSafely)
29122|       return TRUE;
29123|   else
29124|       return FALSE;
29125| }
29126|
29127| BOOL FlushVolume(TCHAR cDriveLetter)
29128| {
29129|     HANDLE hVolume;
29130|     BOOL B;
29131|
29132|     // Open the volume.
29133|     hVolume = OpenVolume(cDriveLetter,GENERIC_WRITE);
29134|     if (hVolume == INVALID_HANDLE_VALUE)
29135|         return FALSE;
29136|
29137|     B = FlushFileBuffers(hVolume);
29138|     //IRP_MJ_FLUSH_BUFFERS
29139|
29140|     // Close the volume so other processes can use the

```

```

    | drive.
29141|     if (!CloseVolume(hVolume))
29142|         return FALSE;
29143|
29144|     return B;
29145| }
29146|
29147| BOOL Win2000_GetDriveAndPartFromDriveLetter(TCHAR
    | cDriveLetter, ULONG *Drive, ULONG *Part)
29148| {
29149|     HANDLE hVolume;
29150|     DWORD dwBytesReturned;
29151|     BOOL B;
29152|     STORAGE_DEVICE_NUMBER sdn;
29153|
29154|     // Open the volume.
29155|     hVolume = OpenVolume(cDriveLetter,0);
29156|     if (hVolume == INVALID_HANDLE_VALUE)
29157|         return FALSE;
29158|
29159|     B = DeviceIoControl( hVolume,
29160|         IOCTL_STORAGE_GET_DEVICE_NUMBER,
29161|         NULL, 0,
29162|         &sdn, sizeof(sdn),
29163|         &dwBytesReturned,
29164|         NULL);
29165|
29166|     if(B) {
29167|         *Drive = sdn.DeviceNumber;
29168|         *Part = sdn.PartitionNumber;
29169|     }
29170|
29171|     // Close the volume so other processes can use the
    | drive.
29172|     if (!CloseVolume(hVolume))
29173|         return FALSE;
29174|
29175|     return B;
29176| }
29177|
29178|
29179| ULONG VDisk_MakeShareEx2( LPWSTR Sharename, WCHAR
    | *DirectoryToShare, tPSM_SecurityInfo *SecurityInfo )
29180| {
29181|     SHARE_INFO_502 si502;
29182|     DWORD dwRes;
29183|     ULONG Err;
29184|     PSID pEveryoneSID = NULL, pAdminSID = NULL,
    | pSystemSID = NULL;
29185|     PACL pACL = NULL;

```

```

29186|  PSECURITY_DESCRIPTOR pSD = NULL;
29187|  EXPLICIT_ACCESS ea[3];
29188|  SID_IDENTIFIER_AUTHORITY SIDAuthWorld =
    | SECURITY_WORLD_SID_AUTHORITY;
29189|  SID_IDENTIFIER_AUTHORITY SIDAuthLocal =
    | SECURITY_LOCAL_SID_AUTHORITY;
29190|  SID_IDENTIFIER_AUTHORITY SIDAuthNT =
    | SECURITY_NT_AUTHORITY;
29191|  SECURITY_ATTRIBUTES sa;
29192|
29193|  if(!SecurityInfo->SecurityDescriptor) {
29194|
29195|      // Create a SID for the BUILTIN\Administrators
    | group.
29196|
29197|      if(! AllocateAndInitializeSid( &SIDAuthNT, 2,
29198|          SECURITY_BUILTIN_DOMAIN_RID,
29199|          DOMAIN_ALIAS_RID_ADMINS,
29200|          0, 0, 0, 0, 0, 0,
29201|          &pAdminSID) ) {
29202|          DLOG((TEXT( "AllocateAndInitializeSid Error
    | %u\n"), GetLastError() ));
29203|          goto Cleanup;
29204|      }
29205|
29206|      // Create a well-known SID for the Everyone
    | group.
29207|
29208|      if(! AllocateAndInitializeSid( &SIDAuthWorld,
    | 1,
29209|          SECURITY_WORLD_RID,
29210|          0, 0, 0, 0, 0, 0, 0, 0,
29211|          &pEveryoneSID) ) {
29212|          DLOG((TEXT( "AllocateAndInitializeSid Error
    | %u\n"), GetLastError() ));
29213|          goto Cleanup;
29214|      }
29215|
29216|      // Create a well-known SID for the system
    | group.
29217|
29218|      if(! AllocateAndInitializeSid( &SIDAuthNT, 1,
29219|          SECURITY_LOCAL_SYSTEM_RID,
29220|          0, 0, 0, 0, 0, 0, 0, 0,
29221|          &pSystemSID) ) {
29222|          DLOG((TEXT( "AllocateAndInitializeSid Error
    | %u\n"), GetLastError() ));
29223|          goto Cleanup;
29224|      }
29225|

```

```

29226|    // Initialize an EXPLICIT_ACCESS structure for
    | an ACE.
29227|    // The ACE will allow the Administrators group
    | full access to the key.
29228|
29229|    ZeroMemory(&ea, 3 * sizeof(EXPLICIT_ACCESS));
29230|    ea[0].grfAccessPermissions =
29231|        GENERIC_READ |
29232|        GENERIC_ALL |
29233|        GENERIC_EXECUTE |
29234|        GENERIC_WRITE |
29235|        SPECIFIC_RIGHTS_ALL |
29236|        STANDARD_RIGHTS_ALL |
29237|        SYNCHRONIZE;
29238|
29239|    ea[0].grfAccessMode = SET_ACCESS;
29240|    ea[0].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
29241|    ea[0].Trustee.TrusteeForm = TRUSTEE_IS_SID;
29242|    ea[0].Trustee.TrusteeType = TRUSTEE_IS_GROUP;
29243|    ea[0].Trustee.ptstrName = (LPTSTR) pAdminSID;
29244|
29245|    ea[1].grfAccessPermissions =
29246|        GENERIC_READ |
29247|        GENERIC_ALL |
29248|        GENERIC_EXECUTE |
29249|        GENERIC_WRITE |
29250|        SPECIFIC_RIGHTS_ALL |
29251|        STANDARD_RIGHTS_ALL |
29252|        SYNCHRONIZE;
29253|
29254|    ea[1].grfAccessMode = SET_ACCESS;
29255|    ea[1].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
29256|    ea[1].Trustee.TrusteeForm = TRUSTEE_IS_SID;
29257|    ea[1].Trustee.TrusteeType = TRUSTEE_IS_USER;
29258|    ea[1].Trustee.ptstrName = (LPTSTR) pSystemSID;
29259|
29260|    ea[2].grfAccessPermissions = GENERIC_READ;
29261|    ea[2].grfAccessMode = SET_ACCESS;
29262|    ea[2].grfInheritance=
    | SUB_CONTAINERS_AND_OBJECTS_INHERIT;
29263|    ea[2].Trustee.TrusteeForm = TRUSTEE_IS_SID;
29264|    ea[2].Trustee.TrusteeType =
    | TRUSTEE_IS_WELL_KNOWN_GROUP;
29265|    ea[2].Trustee.ptstrName = (LPTSTR)
    | pEveryoneSID;
29266|
29267|
29268|    // Create a new ACL that contains the new ACEs.

```

```

29269|
29270|     #define ONLY_ADMIN            1
29271|     #define ADMIN_AND_SYSTEM      2
29272|     #define ADMIN_SYSTEM_AND_EVERYONE 3
29273|
29274|     dwRes =
        | SetEntriesInAcl(ADMIN_SYSTEM_AND_EVERYONE, ea, NULL,
        | &pACL);
29275|     if (ERROR_SUCCESS != dwRes) {
29276|         SetLastError(dwRes);
29277|         DLOG((TEXT( "SetEntriesInAcl Error %u\n"),
        | GetLastError() ));
29278|         goto Cleanup;
29279|     }
29280|
29281|     // Initialize a security descriptor.
29282|
29283|     pSD = (PSECURITY_DESCRIPTOR) LocalAlloc(LPTR,
        | SECURITY_DESCRIPTOR_MIN_LENGTH);
29284|     if (pSD == NULL) {
29285|         DLOG((TEXT( "LocalAlloc Error %u\n"),
        | GetLastError() ));
29286|         goto Cleanup;
29287|     }
29288|
29289|     if (!InitializeSecurityDescriptor(pSD,
        | SECURITY_DESCRIPTOR_REVISION)) {
29290|         DLOG((TEXT( "InitializeSecurityDescriptor
        | Error %u\n"),
29291|             GetLastError() ));
29292|         goto Cleanup;
29293|     }
29294|
29295|     // Add the ACL to the security descriptor.
29296|
29297|     if (!SetSecurityDescriptorDacl(pSD,
29298|         TRUE,    // fDaclPresent flag
29299|         pACL,
29300|         FALSE)) // not a default DACL
29301|     {
29302|         DLOG((TEXT( "SetSecurityDescriptorDacl
        | Error %u\n"), GetLastError() ));
29303|         goto Cleanup;
29304|     }
29305|
29306|     // Initialize a security attributes structure.
29307|
29308|     sa.nLength = sizeof (SECURITY_ATTRIBUTES);
29309|     sa.lpSecurityDescriptor = pSD;
29310|     sa.bInheritHandle = FALSE;

```



```

29311| } // if securitydescriptor
29312|
29313| //
29314| // setup share info structure
29315| //
29316|
29317| // the header says these are TCHARs but they really
    | are WCHARs.
29318| si502.shi502_netname          = Sharename;
29319| si502.shi502_type             =
    | STYPE_DISKTREE;
29320| si502.shi502_remark           =
    | SecurityInfo->Remark;
29321| si502.shi502_permissions      =
    | SecurityInfo->Permissions;
29322| si502.shi502_max_uses         =
    | SecurityInfo->MaxUses;
29323| si502.shi502_current_uses      = 0;
29324| si502.shi502_path             =
    | DirectoryToShare;
29325| si502.shi502_passwd           = NULL;
29326| si502.shi502_reserved         = 0;
29327| si502.shi502_security_descriptor =
    | SecurityInfo->SecurityDescriptor ?
    | SecurityInfo->SecurityDescriptor : pSD;
29328|
29329| DLOG((TEXT("Calling NetShareAdd '%S' for
    | '%S'\n"),Sharename,DirectoryToShare));
29330| Err = NetShareAdd(
29331|     NULL,          // share is on local machine
29332|     502,           // info-level
29333|     (LPBYTE)&si502, // info-buffer
29334|     NULL           // don't bother with parm
29335| );
29336|
29337| if(Err != NO_ERROR) {
29338|     //Err = GetLastError();
29339|     if(Err==NERR_DuplicateShare) {
29340|         // already exists, probally due to crash
29341|         DLOG((TEXT("NetShareAdd Share already
    | exists\n")));
29342|         Err = 0;
29343|     } else {
29344|         DLOG((TEXT("NetShareAdd error!
    | (rc=%08x),LE=%08x\n"), Err,GetLastError()));
29345|         goto Cleanup;
29346|     }
29347| } else {
29348|     DLOG((TEXT("NetShareAdd successful\n")));
29349| }

```

```

29350|
29351|     Err = 0;
29352|
29353| Cleanup:
29354|
29355|     //
29356|     // free allocated resources
29357|     //
29358|     if (pEveryoneSID) {
29359|         FreeSid(pEveryoneSID);
29360|     }
29361|     if (pSystemSID) {
29362|         FreeSid(pSystemSID);
29363|     }
29364|     if (pAdminSID) {
29365|         FreeSid(pAdminSID);
29366|     }
29367|     if (pACL) {
29368|         LocalFree(pACL);
29369|     }
29370|     if (pSD) {
29371|         LocalFree(pSD);
29372|     }
29373|
29374|     return Err;
29375| }
29376|
29377|
29378|
29379| File Listing: VOLUME.h
29380|
29381| // low level volume api
29382| HANDLE OpenVolume(TCHAR cDriveLetter, DWORD
    | dwAccessFlags );
29383| BOOL CloseVolume(HANDLE hVolume);
29384| BOOL LockVolume(HANDLE hVolume);
29385| BOOL UnlockVolume(HANDLE hVolume);
29386| BOOL DismountVolume(HANDLE hVolume);
29387| BOOL PreventRemovalOfVolume(HANDLE hVolume, BOOL
    | fPreventRemoval);
29388| BOOL AutoEjectVolume(HANDLE hVolume);
29389| BOOL EjectVolume(TCHAR cDriveLetter);
29390| BOOL FlushVolume(TCHAR cDriveLetter);
29391|
29392| // ioctls to send to vdisk
29393| ULONG VDisk_UnWriteProtect ( CHAR DriveLetter );
29394| ULONG VDisk_WriteProtect ( CHAR DriveLetter );
29395|
29396| // high level api
29397|

```

```

29398| void FlushAllVolumes();
29399| ULONG VDisk_MakeShare( LPTSTR Sharename, TCHAR Drive );
29400| ULONG VDisk_MakeShareEx( LPWSTR Sharename, WCHAR
    | *DirectoryToShare );
29401| ULONG VDisk_MakeShareEx2( LPWSTR Sharename, WCHAR
    | *DirectoryToShare, tPSM_SecurityInfo *SecurityInfo );
29402| ULONG VDisk_SetSecurity( CHAR Drive );
29403|
29404| ULONG VDisk_AddDriveAndPart( TCHAR OriginalDriveLetter,
    | TCHAR NewDriveLetter, ULONG Drive, ULONG Part,
    | pSnapShot SnapShot );
29405| ULONG VDisk_RemoveDriveAndPart( TCHAR
    | OriginalDriveLetter, TCHAR NewDriveLetter, ULONG Drive,
    | ULONG Part, pSnapShot SnapShot );
29406| BOOL Win2000_GetDriveAndPartFromDriveLetter(TCHAR
    | cDriveLetter, ULONG *Drive, ULONG *Part);
29407|
29408|
29409|
29410| PSM System Driver Source
29411|
29412| The Driver provides kernel mode support for the PSM
    | volume snapshot system. --LPW
29413|
29414|
29415|
29416| File Listing: BIT.cpp
29417|
29418| #include "precomp.h"
29419|
29420| /*-----
    | -----*/
29421| ULONG SafeBitAllocRange ( PFAST_MUTEX Mutex,
    | PRTL_BITMAP bitArray, ULONG Count )
29422| {
29423|     ULONG Ret;
29424|
29425|     pmAcquireMutex ( Mutex, NULL );
29426|     Ret = RtlFindClearBitsAndSet( bitArray, Count, 0 );
29427|     pmReleaseMutex ( Mutex );
29428|     return Ret;
29429| }
29430|
29431| /*-----
    | -----*/
29432| void SafeBitFreeRange ( PFAST_MUTEX Mutex, PRTL_BITMAP
    | bitArray, ULONG Bit, ULONG Count )
29433| {
29434|     pmAcquireMutex ( Mutex, NULL );
29435|     RtlClearBits(bitArray, Bit, Count);

```

```

29436|    pmReleaseMutex ( Mutex );
29437| }
29438|
29439| /*-----
    | -----*/
29440| void SafeBitClear ( PFAST_MUTEX Mutex, PRTL_BITMAP
    | bitArray, ULONG bitNumber )
29441| {
29442|     SafeBitFreeRange( Mutex, bitArray, bitNumber, 1);
29443| }
29444|
29445|
29446| void ChangeTheBitsWithinBounds ( unsigned int ChangeTo,
    | PRTL_BITMAP BMHeader, ULONG Start, ULONG Count)
29447| {
29448|     ASSERT (ChangeTo < 2);
29449|
29450|     if (Start >= BMHeader->SizeOfBitMap) {
29451|         Debug(1,("Ignoring bit action out of map
    | range!\n"));
29452|         return;
29453|     }
29454|
29455|     if (Start+Count > BMHeader->SizeOfBitMap) {
29456|         Debug(1,("Ignoring bit action out of map
    | range!\n"));
29457|         Count= BMHeader->SizeOfBitMap - Start;
29458|     }
29459|
29460|     if (ChangeTo) {
29461|         RtlSetBits ( BMHeader, Start, Count);
29462|     } else {
29463|         RtlClearBits ( BMHeader, Start, Count);
29464|     }
29465| }
29466|
29467| #if 0
29468| #define NoneIn (Start > BM->??)
29469| #define SomeIn (Start+Count>BM->??)
29470|
29471| #define TheSome (Bm->??-Start)
29472| #define TheAll (count)
29473|
29474| #define Warn ( Debug(DEBUG_DCPSM,("Ignoring bit action
    | out of map range!\n")), \
29475|             DbgBreakPoint()
    | \
29476|             )
29477|
29478| #define WithinBoundPartOf(BM,Start,Count) ( NoneIn ?

```

```

    | (Warn,0) : ( Someln ? (Warn, TheSome) : TheAll ))
29479| #endif
29480|
29481|
29482|
29483| File Listing: BIT.h
29484|
29485| #define NUMBER_OF_BITS_IN_A_BYTE 8
29486|
29487| #define CHANGE_TO_CLEAR 0
29488| #define CHANGE_TO_SET 1
29489|
29490| ULONG SafeBitAllocRange ( PFAST_MUTEX Mutex,
    | PRTL_BITMAP bitArray, ULONG Count );
29491| void SafeBitFreeRange ( PFAST_MUTEX Mutex, PRTL_BITMAP
    | bitArray, ULONG Bit, ULONG Count );
29492| void SafeBitClear ( PFAST_MUTEX Mutex, PRTL_BITMAP
    | bitArray, ULONG bitNumber );
29493| void ChangeTheBitsWithinBounds ( unsigned int ChangeTo,
    | PRTL_BITMAP BMHeader, ULONG Start, ULONG Count);
29494|
29495| // The following are convenient debugging functions for
    | bitmaps.
29496|
29497| #ifdef DEBUG
29498|     inline void PsmBitMapValidate ( PRTL_BITMAP BitMap
    | )
29499|     {
29500|         ASSERT (BitMap != NULL);
29501|         if ( BitMap != NULL ) {
29502|             ASSERT (BitMap->Buffer != NULL);
29503|             ASSERT (BitMap->Buffer <
    | &BitMap->SizeOfBitMap || BitMap->Buffer >=
    | PULONG(&BitMap->Buffer + 1));
29504|             ASSERT (BitMap->SizeOfBitMap > 0);
29505|         }
29506|     }
29507|
29508|     inline void PsmBitPositionValidate ( PRTL_BITMAP
    | BitMap, ULONG BitIndex )
29509|     {
29510|         PsmBitMapValidate (BitMap);
29511|         if ( BitMap != NULL ) {
29512|             ASSERT (BitIndex < BitMap->SizeOfBitMap);
29513|         }
29514|     }
29515|
29516|     inline void PsmBitRangeValidate ( PRTL_BITMAP
    | BitMap, ULONG StartIndex, ULONG Count )
29517|     {

```

```

29518|     PsmBitMapValidate (BitMap);
29519|     ASSERT (Count > 0);
29520|     if ( BitMap != NULL ) {
29521|         ASSERT (StartIndex < BitMap->SizeOfBitMap);
29522|         ASSERT (StartIndex+Count <=
| BitMap->SizeOfBitMap);
29523|     }
29524| }
29525| #else
29526| #define PsmBitMapValidate(BitMap)
| ((void)0)
29527| #define PsmBitPositionValidate(BitMap,BitIndex)
| ((void)0)
29528| #define
| PsmBitRangeValidate(BitMap,StartIndex,Count)
| ((void)0)
29529| #endif /*DEBUG*/
29530|
29531|
29532|
29533| File Listing: boot.h
29534|
29535| NTSTATUS InitBootUp(void);
29536| NTSTATUS DelInitBootUp(void);
29537| NTSTATUS BootPSMNotInited( PDEVICE_OBJECT DeviceObject,
| PIRP Irp );
29538|
29539| //#define DO_BOOT_UP
29540|
29541|
29542|
29543| File Listing: BUILDNUM.h
29544|
29545| #define _BuildNumber_ 2100
29546| #define _BuildNumberStr_ "2100\0"
29547| #define _BuildNumberWStr_ L"2100\0"
29548|
29549|
29550|
29551| File Listing: CDP.h
29552|
29553| #define VER_COMPANYNAME_STR    "Columbia Data
| Products, Inc."
29554| #define VER_LEGALTRADEMARKS_STR "PSM\256 is a
| trademark of " VER_COMPANYNAME_STR
29555|
29556| #define VER_LEGALCOPYRIGHT_YEARS "1995-2001"
29557| #define VER_LEGALCOPYRIGHT_STR "Copyright \251 "
| VER_LEGALCOPYRIGHT_YEARS " " VER_COMPANYNAME_STR
29558|

```

```

29559| #if DBG
29560| #define VER_DEBUG          VS_FF_DEBUG
29561| #else
29562| #define VER_DEBUG          0
29563| #endif
29564|
29565| #if BETA
29566| #define VER_PRODUCTBETA_STR  "BETA"
29567| #define VER_PRERELEASE      VS_FF_PRERELEASE
29568| #else
29569| #define VER_PRERELEASE      0
29570| #define VER_PRODUCTBETA_STR  ""
29571| #endif
29572|
29573| #define VER_FILEFLAGSMASK    VS_FFI_FILEFLAGSMASK
29574| #define VER_FILEOS          VOS_NT_WINDOWS32
29575| #define VER_FILEFLAGS      (VER_PRERELEASE |
    | VER_DEBUG)
29576|
29577|
29578|
29579| File Listing: CLEANUP.cpp
29580|
29581| #include "precomp.h"
29582|
29583|
29584| /*-----
    | -----*/
29585| NTSTATUS
29586| PSMANCleanup(
29587|     IN PDEVICE_OBJECT DeviceObject,
29588|     IN PIRP Irp
29589| )
29590|
29591| /*++
29592|
29593| Routine Description:
29594|
29595|     Pass irp to handler
29596|
29597| Arguments:
29598|
29599|     DriverObject - Pointer to device object to being
    | shutdown by system.
29600|     Irp          - IRP involved.
29601|
29602| Return Value:
29603|
29604|     NT Status
29605|

```

```

29606| --*/
29607|
29608| {
29609|     NTSTATUS Status;
29610|
29611|     switch(PsmGetObjectTypes(DeviceObject)) {
29612|         case OBJECT_INTERNAL :
29613|             Status = PSMANCleanupObject(DeviceObject,
29614|                 | Irp);
29615|             break;
29616|         case OBJECT_FILTEREDDISK :
29617|             Status = PSMANCleanupDevice(DeviceObject,
29618|                 | Irp);
29619|             break;
29620|         case OBJECT_VIRTUALDISK :
29621|             Status = PSMANCleanupVdisk(DeviceObject,
29622|                 | Irp);
29623|             break;
29624|         case OBJECT_FS_OBJECT :
29625|             Status = PSMANCleanupFSObject(DeviceObject,
29626|                 | Irp);
29627|             break;
29628|         case OBJECT_FS_FILTER :
29629|             Status = PSMANCleanupFSFilter(DeviceObject,
29630|                 | Irp);
29631|             break;
29632|         default:
29633|             Irp->IoStatus.Status = Status =
29634|                 | STATUS_NO_SUCH_DEVICE;
29635|             Irp->IoStatus.Information = 0 ;
29636|             IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
29637|             break;
29638|     }
29639|     return Status;
29640| } // end PSMANCleanup()
29641|
29642|
29643|
29644|
29645|
29646| /*-----
29647| | -----*/
29648|
29649| STATIC NTSTATUS
29650| PSMANCleanupObject (
29651|     IN PDEVICE_OBJECT DeviceObject,
29652|     IN PIRP Irp
29653| )
29654| /*++
29655|
29656| Routine Description:

```



```

29649|
29650| This routine will match the file object of the
    | cleanup
29651| IRP with any IRP's that are outstanding in our wait
29652| queue. If it finds some, then it will cancel those
29653| IRP's.
29654|
29655|
29656| Arguments:
29657|
29658| DriverObject - Pointer to device object for this
    | operation
29659| Irp          - IRP to work on.
29660|
29661| Return Value:
29662|
29663| NT Status
29664|
29665| --*/
29666| {
29667|
29668|     pOT_USER User;
29669|     NTSTATUS Status=STATUS_SUCCESS;
29670|
29671|     NOT_REFERENCED(DeviceObject);
29672|     PAGED_CODE();
29673|
29674|     /*lint -save -e746 */
29675|     Debug(DEBUG_PROCCALL,("PSManCleanupObject Called:
    | PsGetCurrentProcess=%08x "
29676|         "PsGetCurrentThread=%08x "
29677|         "IoGetCurrentProcess=%08x "
29678|         "KeGetCurrentThread=%08x "
29679|         "User Thread=%08x "
29680|         "OFO=%08x\n",
29681|         PsGetCurrentProcess(),
29682|         PsGetCurrentThread(),
29683|         IoGetCurrentProcess(),
29684|         KeGetCurrentThread(),
29685|         Irp->Tail.Overlay.Thread,
29686|
    | Irp->Tail.Overlay.OriginalFileObject
29687|         ));
29688|     /*lint -restore */
29689|
29690|     // cleanup disptach routine can be called in any
    | thread context, so find
29691|     // object that was created during create
29692|     User = FindPSMUserByFileObject(
    | Irp->Tail.Overlay.OriginalFileObject );

```

```

29693|
29694|     if(User) {
29695|         if(User->Persistent || User->SaveTempOnExit) {
29696|             Status = STATUS_SUCCESS;
29697|             goto ExitClean;
29698|         }
29699|
29700|         pmAcquireMutex ( &PSMUserMutex, NULL );
29701|         // set event in case they are in the middle of
        | enabling psm.
29702|         if(User->AbortEvent) {
29703|             pmSetEvent(User->AbortEvent);
29704|         }
29705|         pmReleaseMutex ( &PSMUserMutex);
29706|     }
29707|
29708|     // Close PSM incase the controlling program
        | disappears (Terminated by user)
29709|     if ((User) && (User->Open)) {
29710|         Debug(DEBUG_INIT,("PSM is still open %d times
        | by user, calling ClosePsm\n",User->Open));
29711|
29712|         while(User->Open) {
29713|
        | if(AcquireOpenCloseResourceOnly(NULL)==STATUS_WAIT_0) {
29714|             InternalClosePSM(User,NULL);
29715|             ReleaseOpenCloseResource();
29716|         }
29717|     }
29718| }
29719|
29720| if(!User) {
29721|     Debug(DEBUG_INIT,("Error User not found during
        | cleanup\n"));
29722| }
29723| // we delete the "user" node during close
29724|
29725| ExitClean:
29726| // Set the information for completion of the
        | IRP_MJ_CLEANUP.
29727| Irp->IoStatus.Status = Status;
29728| Irp->IoStatus.Information = 0 ;
29729|
29730| // Complete the request.
29731| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
29732|
29733| // Return success.
29734|
29735| Debug (DEBUG_PROCCALL,("PSManCleanupObject
        | Done\n")); ;

```

```

29736|   return Status ;
29737| }
29738|
29739|
29740| /*-----
    | -----*/
29741| STATIC NTSTATUS
29742| PSMANCleanupDevice(
29743|   IN PDEVICE_OBJECT DeviceObject,
29744|   IN PIRP Irp
29745|   )
29746|
29747| /*++
29748|
29749| Routine Description:
29750|
29751|   Pass irp to handler
29752|
29753| Arguments:
29754|
29755|   DriverObject - Pointer to device object to being
    | shutdown by system.
29756|   Irp          - IRP involved.
29757|
29758| Return Value:
29759|
29760|   NT Status
29761|
29762| --*/
29763|
29764| {
29765|   NTSTATUS Status;
29766| /*
29767|   PIO_STACK_LOCATION currentIrpStack =
    | IoGetCurrentIrpStackLocation(Irp);
29768|
29769|   TRACE( TRACE_CLEANUP,
29770|
    | currentIrpStack->Parameters.Read.ByteOffset.HighPart,
29771|
    | currentIrpStack->Parameters.Read.ByteOffset.LowPart,
29772|   currentIrpStack->Parameters.Read.Length,
29773|   currentIrpStack->Parameters.Read.Key,
29774|   "");
29775| #ifdef DEBUG
29776|   if(PsmActive) {
29777|     Debug(DEBUG_CLEANUP |
    | DEBUG_PROCCALL,("PSMANCleanupDevice Called Device=%p,
    | Irp=%p\n",DeviceObject,Irp));
29778|   }

```

```

29779| #endif
29780| */
29781|     Status = PSMANPassThru( DeviceObject, Irp );
29782|
29783| /*
29784| #ifdef DEBUG
29785|     if(PsmActive) {
29786|         Debug(DEBUG_CLEANUP |
29787|             | DEBUG_PROCCALL,("PSManCleanupDevice Done  Device=%p,
29788|             | Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
29789|     }
29790| #endif
29791| */
29792|     return Status;
29793| } // end PSMANCleanupDevice()
29794|
29795| /*-----*/
29796| | -----*/
29797|
29798| STATIC NTSTATUS PSMANCleanupVDisk(
29799|     IN PDEVICE_OBJECT DeviceObject,
29800|     IN PIRP Irp
29801| )
29802| {
29803|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
29804|
29805|     Debug(DEBUG_PROCCALL |
29806|         | DEBUG_CLEANUP,("PSManCleanupVDisk Called Dev=%p,
29807|         | Irp=%p\n",DeviceObject,Irp));
29808|     Irp->IoStatus.Information = 0;
29809|     Irp->IoStatus.Status = Status;
29810|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
29811|     Debug(DEBUG_PROCCALL |
29812|         | DEBUG_CLEANUP,("PSManCleanupVDisk Done\n"));
29813|
29814|     return Status;
29815| }
29816|
29817| /*-----*/
29818| | -----*/
29819|
29820| STATIC NTSTATUS
29821| PSMANCleanupFSFilter(
29822|     IN PDEVICE_OBJECT DeviceObject,
29823|     IN PIRP Irp
29824| )
29825| {
29826|     /*++
29827|
29828| Routine Description:
29829|
29830|
29831|
29832|
29833|
29834|
29835|
29836|
29837|
29838|
29839|
29840|
29841|
29842|
29843|
29844|
29845|
29846|
29847|
29848|
29849|
29850|
29851|
29852|
29853|
29854|
29855|
29856|
29857|
29858|
29859|
29860|
29861|
29862|
29863|
29864|
29865|
29866|
29867|
29868|
29869|
29870|
29871|
29872|
29873|
29874|
29875|
29876|
29877|
29878|
29879|
29880|
29881|
29882|
29883|
29884|
29885|
29886|
29887|
29888|
29889|
29890|
29891|
29892|
29893|
29894|
29895|
29896|
29897|
29898|
29899|
29900|
29901|
29902|
29903|
29904|
29905|
29906|
29907|
29908|
29909|
29910|
29911|
29912|
29913|
29914|
29915|
29916|
29917|
29918|
29919|
29920|
29921|
29922|
29923|
29924|
29925|
29926|
29927|
29928|
29929|
29930|
29931|
29932|
29933|
29934|
29935|
29936|
29937|
29938|
29939|
29940|
29941|
29942|
29943|
29944|
29945|
29946|
29947|
29948|
29949|
29950|
29951|
29952|
29953|
29954|
29955|
29956|
29957|
29958|
29959|
29960|
29961|
29962|
29963|
29964|
29965|
29966|
29967|
29968|
29969|
29970|
29971|
29972|
29973|
29974|
29975|
29976|
29977|
29978|
29979|
29980|
29981|
29982|
29983|
29984|
29985|
29986|
29987|
29988|
29989|
29990|
29991|
29992|
29993|
29994|
29995|
29996|
29997|
29998|
29999|
30000|

```

```

29822|   Pass irp to handler
29823|
29824| Arguments:
29825|
29826|   DriverObject - Pointer to device object to being
      | shutdown by system.
29827|   Irp          - IRP involved.
29828|
29829| Return Value:
29830|
29831|   NT Status
29832|
29833| --*/
29834|
29835| {
29836|   NTSTATUS Status;
29837| /*
29838| #ifdef DEBUG
29839|   if(PsmActive) {
29840|     Debug(DEBUG_CLEANUP |
      | DEBUG_PROCCALL,("PSManCleanupFSFilter Called Device=%p,
      | Irp=%p\n",DeviceObject,Irp));
29841|   }
29842| #endif
29843| */
29844|   Status = PSManFSPassThru( DeviceObject, Irp );
29845| /*
29846| #ifdef DEBUG
29847|   if(PsmActive) {
29848|     Debug(DEBUG_CLEANUP |
      | DEBUG_PROCCALL,("PSManCleanupFSFilter Done   Device=%p,
      | Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
29849|   }
29850| #endif
29851| */
29852|   return Status;
29853| } // end PSManCleanupFSFilter()
29854|
29855| /*-----
      | -----*/
29856| STATIC NTSTATUS PSManCleanupFSObject(
29857|   IN PDEVICE_OBJECT DeviceObject,
29858|   IN PIRP Irp
29859| )
29860| {
29861|   NTSTATUS Status=STATUS_SUCCESS;
29862|
29863|   Debug(DEBUG_PROCCALL |
      | DEBUG_CLEANUP,("PSManCleanupFSObject Called Dev=%p,
      | Irp=%p\n",DeviceObject,Irp));

```

```
29864| Irp->IoStatus.Information = 0;
29865| Irp->IoStatus.Status = Status;
29866| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
29867| Debug(DEBUG_PROCCALL |
    | DEBUG_CLEANUP,("PSManCleanupFSObject Done\n"));
29868|
29869| return Status;
29870| }
29871|
29872|
29873|
29874| File Listing: CLEANUP.h
29875|
29876| NTSTATUS
29877| PSManCleanup(
29878|     IN PDEVICE_OBJECT DeviceObject,
29879|     IN PIRP Irp
29880| );
29881|
29882| STATIC void PSManCancelRoutine ( IN PIRP Irp );
29883|
29884| STATIC void
29885| PSManCancelObject(
29886|     IN PDEVICE_OBJECT DeviceObject,
29887|     IN PIRP Irp
29888| );
29889|
29890| STATIC NTSTATUS
29891| PSManCleanupObject (
29892|     IN PDEVICE_OBJECT DeviceObject,
29893|     IN PIRP Irp
29894| );
29895|
29896| STATIC NTSTATUS
29897| PSManCleanupDevice(
29898|     IN PDEVICE_OBJECT DeviceObject,
29899|     IN PIRP Irp
29900| );
29901|
29902| STATIC NTSTATUS PSManCleanupVDisk(
29903|     IN PDEVICE_OBJECT DeviceObject,
29904|     IN PIRP Irp
29905| );
29906|
29907| STATIC NTSTATUS PSManCleanupFSObject(
29908|     IN PDEVICE_OBJECT DeviceObject,
29909|     IN PIRP Irp
29910| );
29911|
29912| STATIC NTSTATUS PSManCleanupFSFilter(
```

```

29913|    IN PDEVICE_OBJECT DeviceObject,
29914|    IN PIRP Irp
29915| );
29916|
29917|
29918|
29919| File Listing: CLOSE.cpp
29920|
29921| #include "precomp.h"
29922|
29923|
29924| /*-----
    | -----*/
29925| NTSTATUS
29926| PSMANClose(
29927|     IN PDEVICE_OBJECT DeviceObject,
29928|     IN PIRP Irp
29929| )
29930|
29931| /*++
29932|
29933| Routine Description:
29934|
29935|     Pass irp to handler
29936|
29937| Arguments:
29938|
29939|     DriverObject - Pointer to device object to being
    | closed
29940|     Irp          - IRP involved.
29941|
29942| Return Value:
29943|
29944|     NT Status
29945|
29946| --*/
29947|
29948| {
29949|     NTSTATUS Status;
29950|
29951|     switch(PsmGetObjectTypes(DeviceObject)) {
29952|     case OBJECT_INTERNAL :
29953|         Status = PSMANCloseObject(DeviceObject,
    | Irp);
29954|         break;
29955|     case OBJECT_FILTEREDDISK :
29956|         Status = PSMANCloseDevice(DeviceObject,
    | Irp);
29957|         break;
29958|     case OBJECT_VIRTUALDISK :

```

```

29959|         Status = PSMANCloseVDisk(DeviceObject,
    | Irp);
29960|         break;
29961|     case OBJECT_FS_FILTER :
29962|         Status = PSMANCloseFSFilter(DeviceObject,
    | Irp);
29963|         break;
29964|     case OBJECT_FS_OBJECT :
29965|         Status = PSMANCloseFSObject(DeviceObject,
    | Irp);
29966|         break;
29967|     default:
29968|         Irp->IoStatus.Status = Status =
    | STATUS_NO_SUCH_DEVICE;
29969|         Irp->IoStatus.Information = 0 ;
29970|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
29971|         break;
29972|     }
29973|
29974|     return Status;
29975|
29976| } // end PSMANClose()
29977|
29978|
29979| STATIC NTSTATUS
29980| PSMANCloseObject(
29981|     IN PDEVICE_OBJECT DeviceObject,
29982|     IN PIRP Irp
29983| )
29984|
29985| /*++
29986|
29987| Routine Description:
29988|
29989|     This routine is called when a user mode apps closes
    | the handle, or
29990|     is unloaded.
29991|
29992| Arguments:
29993|
29994|     DeviceObject - Pointer to device object being
    | closed.
29995|     Irp          - IRP involved.
29996|
29997| Return Value:
29998|
29999|     NT Status
30000|
30001| --*/
30002|

```



```

30003| {
30004|     pOT_USER User;
30005|
30006|     PAGED_CODE();
30007|
30008|     /*lint -save -e746 */
30009|     Debug(DEBUG_PROCCALL,("PSManCloseObject Called:
        | PsGetCurrentProcess=%08x "
30010|         "PsGetCurrentThread=%08x "
30011|         "IoGetCurrentProcess=%08x "
30012|         "KeGetCurrentThread=%08x "
30013|         "User Thread=%08x "
30014|         "OFO=%08x\n",
30015|         PsGetCurrentProcess(),
30016|         PsGetCurrentThread(),
30017|         IoGetCurrentProcess(),
30018|         KeGetCurrentThread(),
30019|         Irp->Tail.Overlay.Thread,
30020|
        | Irp->Tail.Overlay.OriginalFileObject
30021|         ));
30022|     /*lint -restore */
30023|
30024|     // close disptach routine can be called in any
        | thread context, so find
30025|     // object that was created during create
30026|     User = FindPSMUserByFileObject(
        | Irp->Tail.Overlay.OriginalFileObject );
30027|
30028|     // Close PSM incase the controlling program
        | disappears (Terminated by user)
30029|     // this should never occur as it occurs in
        | cleanup.c but just in case..
30030|
30031|     if((User) && (User->Persistent ||
        | User->SaveTempOnExit)) {
30032|         goto ExitClose;
30033|     }
30034|
30035|     if ((User) && (User->Open)) {
30036|         Debug(DEBUG_INIT,("PSM is still open %d times
        | by user, calling ClosePsm\n",User->Open));
30037|         while(User->Open) {
30038|
            | if(AcquireOpenCloseResourceOnly(NULL)==STATUS_WAIT_0) {
30039|                 InternalClosePSM(User,NULL);
30040|                 ReleaseOpenCloseResource();
30041|             }
30042|         }
30043|     }

```

```

30044|
30045|     if(User) {
30046|         DeletePSMUser(User);
30047|         FREE_POINTER(User);
30048|     } else {
30049|         Debug(DEBUG_INIT,("Error User not found during
| close\n"));
30050|     }
30051|
30052|     // cant access GlobalData->NumActive here as if a
| close request comes in after
30053|     // an init, but before active, then we close psm
| inadvertently and cause a bsod.
30054|     // we do not want to wait on the global open/close
| event as it would take to long
30055|     // if an open was pending.
30056| //     if((GlobalData->NumActive==0) &&
| (PSManPSMInitied)) {
30057|
30058|     // so now check to see if no users left and psm
| still initied.
30059|     pmAcquireMutex ( &PSMUserMutex, NULL );
30060|     User = GlobalData->PSMUsers;
30061|     pmReleaseMutex ( &PSMUserMutex );
30062|
30063|     if((User==NULL) && (PSManPSMInitied)) {
30064|         // no users left, but psm is initied,
| application probably exited during Psm_Enable
30065|         Debug(DEBUG_CLOSE,("PSManCloseObject: Psm
| initied, but not active, closing\n"));
30066|
| if(AcquireOpenCloseResourceOnly(NULL)==STATUS_WAIT_0) {
30067|             InternalClosePSM(User,NULL);
30068|             ReleaseOpenCloseResource();
30069|         }
30070|     }
30071|
30072| ExitClose:
30073|     Irp->IoStatus.Status = STATUS_SUCCESS;
30074|     Irp->IoStatus.Information = 0;
30075|     IoCompleteRequest(Irp, IO_NO_INCREMENT);
30076|
30077|     Debug(DEBUG_PROCCALL,("PSManCloseObject Done\n"));
30078|     return STATUS_SUCCESS;
30079|
30080| } // end PSManCloseObject()
30081|
30082|
30083| /*-----
| -----*/

```

```

30084| STATIC NTSTATUS
30085| PSMANCloseDevice(
30086|     IN PDEVICE_OBJECT DeviceObject,
30087|     IN PIRP Irp
30088| )
30089|
30090| /*++
30091|
30092| Routine Description:
30093|
30094|     Pass irp to handler
30095|
30096| Arguments:
30097|
30098|     DriverObject - Pointer to device object to being
        | shutdown by system.
30099|     Irp         - IRP involved.
30100|
30101| Return Value:
30102|
30103|     NT Status
30104|
30105| --*/
30106|
30107| {
30108|     NTSTATUS Status;
30109| /*
30110|     PIO_STACK_LOCATION currentIrpStack =
        | IoGetCurrentIrpStackLocation(Irp);
30111|     TRACE( TRACE_CLOSE,
30112|
        | currentIrpStack->Parameters.Read.ByteOffset.HighPart,
30113|
        | currentIrpStack->Parameters.Read.ByteOffset.LowPart,
30114|         currentIrpStack->Parameters.Read.Length,
30115|         currentIrpStack->Parameters.Read.Key,
30116|         "");
30117| #ifdef DEBUG
30118|     if(PsmActive) {
30119|         Debug(DEBUG_CLOSE |
        | DEBUG_PROCCALL,("PSMANCloseDevice Called Device=%p,
        | Irp=%p\n",DeviceObject,Irp));
30120|     }
30121| #endif
30122| */
30123|
30124|     Status = PSMANPassThru( DeviceObject, Irp );
30125|
30126| /*
30127| #ifdef DEBUG

```

```

30128|   if(PsmActive) {
30129|       Debug(DEBUG_CLOSE |
| DEBUG_PROCCALL,("PManCloseDevice Done   Device=%p,
| Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
30130|   }
30131| #endif
30132| */
30133|   return Status;
30134| } // end PManCloseDevice()
30135|
30136| /*-----
| -----*/
30137| STATIC NTSTATUS PManCloseVDisk(
30138|   IN PDEVICE_OBJECT DeviceObject,
30139|   IN PIRP Irp
30140| )
30141| {
30142|   NTSTATUS Status=STATUS_SUCCESS;
30143|
30144|   Debug(DEBUG_PROCCALL |
| DEBUG_CLOSE,("PmanCloseVDisk Called Dev=%p,
| Irp=%p\n",DeviceObject,Irp));
30145|   Irp->IoStatus.Information = 0;
30146|   Irp->IoStatus.Status = Status;
30147|
30148|   IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
30149|   Debug(DEBUG_PROCCALL |
| DEBUG_CLOSE,("PmanCloseVDisk Done\n"));
30150|
30151|   return Status;
30152| }
30153|
30154| /*-----
| -----*/
30155| STATIC NTSTATUS
30156| PManCloseFSFilter(
30157|   IN PDEVICE_OBJECT DeviceObject,
30158|   IN PIRP Irp
30159| )
30160|
30161| /*++
30162|
30163| Routine Description:
30164|
30165|   Pass irp to handler
30166|
30167| Arguments:
30168|
30169|   DriverObject - Pointer to device object to being
| shutdown by system.

```

```

30170|    Irp        - IRP involved.
30171|
30172| Return Value:
30173|
30174|    NT Status
30175|
30176| --*/
30177|
30178| {
30179|     return PSMANFSClose(DeviceObject,Irp);
30180| } // end PSMANCloseFSFilter()
30181|
30182| /*-----
    | -----*/
30183| STATIC NTSTATUS PSMANCloseFSObject(
30184|     IN PDEVICE_OBJECT DeviceObject,
30185|     IN PIRP Irp
30186| )
30187| {
30188|     NTSTATUS Status=STATUS_SUCCESS;
30189|
30190|     Debug(DEBUG_PROCCALL |
    | DEBUG_CLOSE,("PsmanCloseFSObject Called Dev=%p,
    | Irp=%p\n",DeviceObject,Irp));
30191|     Irp->IoStatus.Information = 0;
30192|     Irp->IoStatus.Status = Status;
30193|
30194|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
30195|     Debug(DEBUG_PROCCALL |
    | DEBUG_CLOSE,("PsmanCloseFSObject Done\n"));
30196|
30197|     return Status;
30198| }
30199|
30200|
30201|
30202| File Listing: CLOSE.h
30203|
30204| NTSTATUS
30205| PSMANClose(
30206|     IN PDEVICE_OBJECT DeviceObject,
30207|     IN PIRP Irp
30208| );
30209|
30210| STATIC NTSTATUS
30211| PSMANCloseObject(
30212|     IN PDEVICE_OBJECT DeviceObject,
30213|     IN PIRP Irp
30214| );
30215|

```

```

30216| STATIC NTSTATUS
30217| PSMAN_CLOSE_DEVICE(
30218|     IN PDEVICE_OBJECT DeviceObject,
30219|     IN PIRP Irp
30220| );
30221| STATIC NTSTATUS PSMAN_CLOSE_VDISK(
30222|     IN PDEVICE_OBJECT DeviceObject,
30223|     IN PIRP Irp
30224| );
30225| STATIC NTSTATUS PSMAN_CLOSE_FS_OBJECT(
30226|     IN PDEVICE_OBJECT DeviceObject,
30227|     IN PIRP Irp
30228| );
30229| STATIC NTSTATUS PSMAN_CLOSE_FS_FILTER(
30230|     IN PDEVICE_OBJECT DeviceObject,
30231|     IN PIRP Irp
30232| );
30233|
30234|
30235|
30236| File Listing: CONT.cpp
30237|
30238| #include "precomp.h"
30239|
30240|
30241| #if TRACK_CONTENTIONS
30242| KSPIN_LOCK ContentionSpinLock=0;
30243| LIST_ENTRY
    | RegisteredObjects={&RegisteredObjects,&RegisteredObjects
    | };
30244|
30245|
30246| typedef struct sRegisteredObject {
30247|     LIST_ENTRY ListEntry;
30248|     char      Name[40];
30249|     PVOID     Object;
30250|     tPrimateObjectType ObjectType;
30251|     tpmContentionCounters Counter;
30252| } tRegisteredObject,*pRegisteredObject;
30253|
30254|
30255| STATIC pRegisteredObject FindObject ( PVOID Object )
30256| {
30257|     PLIST_ENTRY ListEntry;
30258|     KIRQL OldIrql = 0;
30259|
30260|     KeAcquireSpinLock ( &ContentionSpinLock, &OldIrql
    | );
30261|
30262|     ListEntry = RegisteredObjects.Flink;

```

```

30263|
30264| while(ListEntry!=&RegisteredObjects) {
30265|     pRegisteredObject ro =
        | CONTAINING_RECORD(ListEntry,tRegisteredObject,ListEntry)
        | ;
30266|     if(ro->Object == Object ) {
30267|         KeReleaseSpinLock ( &ContentionSpinLock,
            | OldIrql );
30268|         return ro;
30269|     }
30270|     ListEntry = ListEntry->Flink;
30271| }
30272| KeReleaseSpinLock ( &ContentionSpinLock, OldIrql );
30273| return NULL;
30274| }
30275|
30276|
30277| NTSTATUS pmRegisterObject( PVOID Object, char *Name,
        | tPrimateObjectType ObjectType )
30278| {
30279|     KIRQL OldIrql = 0;
30280|     pRegisteredObject ro =
        | ExAllocatePool(NonPagedPool,sizeof(tRegisteredObject));
30281|     if(ro) {
30282|         memset(ro,sizeof(tRegisteredObject),0);
30283|         ro->Object = Object;
30284|         ro->ObjectType = ObjectType;
30285|         strcpy(ro->Name,Name);
30286|
30287|         KeAcquireSpinLock ( &ContentionSpinLock,
            | &OldIrql );
30288|
        | InsertTailList(&RegisteredObjects,&ro->ListEntry);
30289|         KeReleaseSpinLock ( &ContentionSpinLock,
            | OldIrql );
30290|     } else {
30291|         return STATUS_INSUFFICIENT_RESOURCES;
30292|     }
30293|
30294|     return STATUS_SUCCESS;
30295| }
30296|
30297|
30298| NTSTATUS pmDeRegisterObject( PVOID Object )
30299| {
30300|     pRegisteredObject ro=FindObject(Object);
30301|     KIRQL OldIrql;
30302|     if(ro) {
30303|         KeAcquireSpinLock ( &ContentionSpinLock,
            | &OldIrql );

```

```

30304|    RemoveEntryList(&ro->ListEntry);
30305|    KeReleaseSpinLock ( &ContentionSpinLock,
    | OldIrql );
30306|    ExFreePool(ro);
30307|    return STATUS_SUCCESS;
30308| } else {
30309|     DbgBreakPoint();
30310|     return STATUS_INVALID_PARAMETER;
30311| }
30312| }
30313|
30314|
30315| NTSTATUS pmWaitForMultipleObjects( PVOID Objects[],
    | ULONG Num, PLARGE_INTEGER Timeout )
30316| {
30317|     ULONG i;
30318|     pRegisteredObject ro;
30319|     NTSTATUS Err;
30320|
30321|     ASSERT(Timeout==NULL);
30322|
30323|     // Objects valid for this call
30324|     // Semaphore
30325|     // Event
30326|     // Not valid:
30327|     // Mutex   - Needs special function to acquire
30328|     // RwLock  - Needs special function to acquire
30329|     // SpinLock - Needs special function to acquire
30330|
30331|
30332|     for(i=0;i<Num;i++) {
30333|         ro = FindObject(Objects[i]);
30334|         if(ro) {
30335|             // this is a cheat
30336|             // we only wait for the first semaphore to
    | occur
30337|             ASSERT(ro->ObjectType==pmSemaphore);
30338|             return (Err =
    | pmAcquireSemaphore(Objects[i],Timeout))==0 ? i : Err;
30339|         }
30340|     }
30341|     // no semaphores so just wait
30342|     return
    | KeWaitForMultipleObjects(Num,Objects,WaitAny,Suspended,(
    | KPROCESSOR_MODE)KernelMode,FALSE,Timeout,NULL);
30343| }
30344|
30345| //-----
    | -----
30346| //  Statistics Collection Functions

```



```

30347| //-----
      | -----
30348|
30349|
30350| NTSTATUS pmAcquireSemaphore (
30351|     PKSEMAPHORE Semaphore,
30352|     PLARGE_INTEGER Timeout )
30353| {
30354|     // Check to see if we will block... and remember
      | for later.
30355|     LONG semState = KeReadStateSemaphore ( Semaphore );
30356|     BOOLEAN willProbablyBlock = (semState <= 0);
30357|
30358|     LARGE_INTEGER performanceFrequency = {0};
30359|     LARGE_INTEGER timeBefore = {0};
30360|     NTSTATUS ntStatus = 0;
30361|     KIRQL OldIrql = 0;
30362|     pRegisteredObject ro = FindObject(Semaphore);
30363|
30364|     if(ro) {
30365|         if ( willProbablyBlock ) {
30366|             // Don't call a potentially expensive
      | routine unless we think we're going to block.
30367|             // What time is it before acquiring the
      | semaphore?
30368|             timeBefore = KeQueryPerformanceCounter (
      | &performanceFrequency );
30369|         }
30370|
30371|         // acquire semaphore
30372|         ntStatus = KeWaitForSingleObject (
30373|             Semaphore,
30374|             Executive,
30375|             KernelMode,
30376|             FALSE,
30377|             Timeout );
30378|
30379|         KeAcquireSpinLock ( &ContentionSpinLock,
      | &OldIrql );
30380|         ++ro->Counter.SemaphoreCounter.Total.QuadPart;
30381|         KeReleaseSpinLock ( &ContentionSpinLock,
      | OldIrql );
30382|
30383|         if ( willProbablyBlock ) {
30384|             // Getting here means we (almost) certainly
      | blocked
30385|             // while waiting for the semaphore.
30386|
30387|             // What time is it after acquiring the
      | semaphore?

```

```

30388|         LARGE_INTEGER timeAfter =
    | KeQueryPerformanceCounter ( &performanceFrequency );
30389|
30390|         ULARGE_INTEGER elapsed = {0};
30391|         elapsed.QuadPart = timeAfter.QuadPart -
    | timeBefore.QuadPart;
30392|
30393|         KeAcquireSpinLock ( &ContentionSpinLock,
    | &OldIrql );
30394|
    | ro->Counter.SemaphoreCounter.TotalWaitTime.QuadPart +=
    | elapsed.QuadPart;
30395|         if ( elapsed.QuadPart >
    | ro->Counter.SemaphoreCounter.MaxWaitTime.QuadPart ) {
30396|
    | ro->Counter.SemaphoreCounter.MaxWaitTime.QuadPart =
    | elapsed.QuadPart;
30397|         }
30398|
    | ++ro->Counter.SemaphoreCounter.Contentions.QuadPart;
30399|         KeReleaseSpinLock ( &ContentionSpinLock,
    | OldIrql );
30400|     }
30401| } else {
30402|     Debug(DEBUG_INFO,("Invalid semaphore object
    | %08x\n",Semaphore));
30403|     DbgBreakPoint();
30404| }
30405|
30406| return ntStatus;
30407| }
30408|
30409|
30410| void pmAcquireMutex (
30411|     PFAST_MUTEX FastMutex,
30412|     PLARGE_INTEGER Timeout)
30413| {
30414|     KIRQL OldIrql = 0;
30415|     LARGE_INTEGER performanceFrequency = {0};
30416|     pRegisteredObject ro = FindObject(FastMutex);
30417|
30418|     if ( ro ) {
30419|         if ( ExTryToAcquireFastMutex(FastMutex) ) {
30420|             // Getting here means we did acquire the
    | mutex without wait.
30421|             KeAcquireSpinLock ( &ContentionSpinLock,
    | &OldIrql );
30422|             ++ro->Counter.MutexCounter.Total.QuadPart;
30423|             KeReleaseSpinLock ( &ContentionSpinLock,
    | OldIrql );

```

```

30424|
30425|     } else {
30426|         // Getting here means we will have to wait
        | before acquiring the mutex.
30427|
30428|         ULARGE_INTEGER elapsed = {0};
30429|         LARGE_INTEGER timeAfter = {0};
30430|         LARGE_INTEGER timeBefore =
        | KeQueryPerformanceCounter ( &performanceFrequency );
30431|         ExAcquireFastMutex ( FastMutex );
30432|         timeAfter = KeQueryPerformanceCounter (
        | &performanceFrequency );
30433|         elapsed.QuadPart = timeAfter.QuadPart -
        | timeBefore.QuadPart;
30434|
30435|         KeAcquireSpinLock ( &ContentionSpinLock,
        | &OldIrql ); // hold your breath!
30436|
30437|         ++ro->Counter.MutexCounter.Total.QuadPart;
30438|
        | ++ro->Counter.MutexCounter.Contentions.QuadPart;
30439|
        | ro->Counter.MutexCounter.TotalWaitTime.QuadPart +=
        | elapsed.QuadPart;
30440|         if ( elapsed.QuadPart >
        | ro->Counter.MutexCounter.MaxWaitTime.QuadPart ) {
30441|
        | ro->Counter.MutexCounter.MaxWaitTime.QuadPart =
        | elapsed.QuadPart;
30442|         }
30443|
30444|         KeReleaseSpinLock ( &ContentionSpinLock,
        | OldIrql ); // now exhale... ah!
30445|     }
30446| } else {
30447|     Debug(DEBUG_INFO,("Invalid mutex object
        | %08x\n",FastMutex));
30448|     DbgBreakPoint();
30449| }
30450| }
30451|
30452|
30453| void pmAcquireSpinLock (
30454|     PKSPIN_LOCK SpinLock,
30455|     PKIRQL callersOldIrql)
30456| {
30457|     ULARGE_INTEGER elapsed = {0};
30458|     LARGE_INTEGER timeAfter = {0};
30459|     LARGE_INTEGER timeBefore = {0};
30460|     LARGE_INTEGER performanceFrequency = {0};

```

```

30461|   pRegisteredObject ro = FindObject(SpinLock);
30462|
30463|   if ( ro ) {
30464|       KIRQL OldIrql = 0;
30465|       BOOLEAN WillBlock = (*SpinLock != 0);
30466|
30467|       if ( WillBlock ) {
30468|           timeBefore = KeQueryPerformanceCounter (
30469|               | &performanceFrequency );
30470|       }
30471|       KeAcquireSpinLock ( SpinLock, callersOldIrql );
30472|       | // This is for the caller
30473|       if ( WillBlock ) {
30474|           timeAfter = KeQueryPerformanceCounter (
30475|               | &performanceFrequency );
30476|           elapsed.QuadPart = timeAfter.QuadPart -
30477|               | timeBefore.QuadPart;
30478|       }
30479|       KeAcquireSpinLock ( &ContentionSpinLock,
30480|           | &OldIrql );
30481|       ++ro->Counter.SpinLockCounter.Total.QuadPart;
30482|       if ( WillBlock ) {
30483|           | ++ro->Counter.SpinLockCounter.Contentions.QuadPart;
30484|           if ( elapsed.QuadPart >
30485|               | ro->Counter.SpinLockCounter.MaxWaitTime.QuadPart ) {
30486|               | ro->Counter.SpinLockCounter.MaxWaitTime.QuadPart =
30487|                   | elapsed.QuadPart;
30488|           }
30489|           | ro->Counter.SpinLockCounter.TotalWaitTime.QuadPart +=
30490|               | elapsed.QuadPart;
30491|       }
30492|       KeReleaseSpinLock ( &ContentionSpinLock,
30493|           | OldIrql );
30494|   } else {
30495|       Debug(DEBUG_INFO,("Invalid mutex object
30496|           | %08x\n",SpinLock));
30497|       DbgBreakPoint();
30498|   }
30499| }
30500| }
30501| }

```

```

30498|
30499| BOOLEAN pmAcquireReaderLock (
30500|     PERESOURCE Resource,
30501|     BOOLEAN Wait)
30502| {
30503|     BOOLEAN Immediate = ExAcquireResourceSharedLite (
        | Resource, FALSE );
30504|     BOOLEAN ReturnValue = Immediate;
30505|     ULARGE_INTEGER elapsed = {0};
30506|     LARGE_INTEGER timeAfter = {0};
30507|     LARGE_INTEGER timeBefore = {0};
30508|     LARGE_INTEGER performanceFrequency = {0};
30509|     ULONG NumActiveReaders = 0;
30510|     KIRQL OldIrql = 0;
30511|     pRegisteredObject ro = FindObject(Resource);
30512|
30513|     if ( ro ) {
30514|
30515|         KeAcquireSpinLock ( &ContentionSpinLock,
            | &OldIrql );
30516|         ++ro->Counter.RwCounter.TotalReaders.QuadPart;
30517|         KeReleaseSpinLock ( &ContentionSpinLock,
            | OldIrql );
30518|
30519|         if ( !Immediate ) {
30520|             timeBefore = KeQueryPerformanceCounter (
                | &performanceFrequency );
30521|             ReturnValue = ExAcquireResourceSharedLite (
                | Resource, TRUE );
30522|             NumActiveReaders = ExGetSharedWaiterCount (
                | Resource );
30523|
30524|             timeAfter = KeQueryPerformanceCounter (
                | &performanceFrequency );
30525|             elapsed.QuadPart = timeAfter.QuadPart -
                | timeBefore.QuadPart;
30526|
30527|             KeAcquireSpinLock ( &ContentionSpinLock,
                | &OldIrql );
30528|
                | ro->Counter.RwCounter.TotalReadWaitTime.QuadPart +=
                | elapsed.QuadPart;
30529|             if ( elapsed.QuadPart >
                | ro->Counter.RwCounter.MaxReadWaitTime.QuadPart ) {
30530|
                | ro->Counter.RwCounter.MaxReadWaitTime.QuadPart =
                | elapsed.QuadPart;
30531|             }
30532|
                | ++ro->Counter.RwCounter.ReaderContentions.QuadPart;

```

```

30533|
    | ro->Counter.RwCounter.TotalActiveReaders.QuadPart +=
    | NumActiveReaders;
30534|         if(NumActiveReaders >
    | ro->Counter.RwCounter.MaxReaders.LowPart) {
30535|
    | ro->Counter.RwCounter.MaxReaders.LowPart =
    | NumActiveReaders;
30536|         }
30537|         KeReleaseSpinLock ( &ContentionSpinLock,
    | OldIrql );
30538|     } else {
30539|         NumActiveReaders = ExGetSharedWaiterCount (
    | Resource );
30540|         KeAcquireSpinLock ( &ContentionSpinLock,
    | &OldIrql );
30541|
    | ro->Counter.RwCounter.TotalActiveReaders.QuadPart +=
    | NumActiveReaders;
30542|         if(NumActiveReaders >
    | ro->Counter.RwCounter.MaxReaders.LowPart) {
30543|
    | ro->Counter.RwCounter.MaxReaders.LowPart =
    | NumActiveReaders;
30544|         }
30545|         KeReleaseSpinLock ( &ContentionSpinLock,
    | OldIrql );
30546|     }
30547| } else {
30548|     Debug(DEBUG_INFO,("Invalid resource object
    | %08x\n",Resource));
30549|     DbgBreakPoint();
30550| }
30551|
30552| return ReturnValue;
30553| }
30554|
30555|
30556| BOOLEAN pmAcquireWriterLock (
30557|     PERESOURCE Resource,
30558|     BOOLEAN Wait)
30559| {
30560|     BOOLEAN Immediate = ExAcquireResourceExclusiveLite
    | ( Resource, FALSE );
30561|     BOOLEAN ReturnValue = Immediate;
30562|     ULARGE_INTEGER elapsed = {0};
30563|     LARGE_INTEGER timeAfter = {0};
30564|     LARGE_INTEGER timeBefore = {0};
30565|     LARGE_INTEGER performanceFrequency = {0};
30566|     KIRQL OldIrql = 0;

```

```

30567|   pRegisteredObject ro = FindObject(Resource);
30568|
30569|   if ( ro ) {
30570|
30571|       KeAcquireSpinLock ( &ContentionSpinLock,
        | &OldIrql );
30572|       ++ro->Counter.RwCounter.TotalWriters.QuadPart;
30573|       KeReleaseSpinLock ( &ContentionSpinLock,
        | OldIrql );
30574|
30575|       if ( !Immediate ) {
30576|           timeBefore = KeQueryPerformanceCounter (
        | &performanceFrequency );
30577|           ReturnValue =
        | ExAcquireResourceExclusiveLite ( Resource, TRUE );
30578|           timeAfter = KeQueryPerformanceCounter (
        | &performanceFrequency );
30579|
30580|           elapsed.QuadPart = timeAfter.QuadPart -
        | timeBefore.QuadPart;
30581|
30582|           KeAcquireSpinLock ( &ContentionSpinLock,
        | &OldIrql );
30583|
        | ro->Counter.RwCounter.TotalWriteWaitTime.QuadPart +=
        | elapsed.QuadPart;
30584|           if ( elapsed.QuadPart >
        | ro->Counter.RwCounter.MaxWriteWaitTime.QuadPart ) {
30585|
        | ro->Counter.RwCounter.MaxWriteWaitTime.QuadPart =
        | elapsed.QuadPart;
30586|           }
30587|
        | ++ro->Counter.RwCounter.WriterContentions.QuadPart;
30588|           KeReleaseSpinLock ( &ContentionSpinLock,
        | OldIrql );
30589|       }
30590|
30591|   } else {
30592|       Debug(DEBUG_INFO,("Invalid resource object
        | %08x\n",Resource));
30593|       DbgBreakPoint();
30594|   }
30595|
30596|   return ReturnValue;
30597| }
30598|
30599|
30600| //-----
        | -----

```

```

30601| // Statistics Dump Functions
30602| //-----
30603|
30604|
30605| void pmDump_MutexCounters (
30606|     const pRegisteredObject Ro)
30607| {
30608|     ULARGE_INTEGER AverageWaitTime = {0};
30609|     tMutexCounters *Counters=&Ro->Counter.MutexCounter;
30610|
30611|     if ( Counters->Contentions.QuadPart > 0 ) {
30612|         AverageWaitTime.QuadPart =
30613|             Counters->TotalWaitTime.QuadPart /
30614|             | Counters->Contentions.QuadPart;
30615|     }
30616|
30617|     Debug(DEBUG_INFO,("Mutex   : %08x
30618|         | '%s'\n",Ro->Object,Ro->Name));
30619|     Debug(DEBUG_INFO,("         Total=%l64d,
30620|         | Cont=%l64d, WaitT=%l64d, MaxW=%l64d\n",
30621|         Counters->Total,
30622|         Counters->Contentions,
30623|         Counters->TotalWaitTime,
30624|         Counters->MaxWaitTime));
30625|     Debug(DEBUG_INFO,("
30626|         | AverageW=%l64d\n",AverageWaitTime));
30627| }
30628|
30629| void pmDump_SemaphoreCounters (
30630|     const pRegisteredObject Ro)
30631| {
30632|     ULARGE_INTEGER AverageWaitTime = {0};
30633|     tSemaphoreCounters
30634|         | *Counters=&Ro->Counter.SemaphoreCounter;
30635|
30636|     if ( Counters->Contentions.QuadPart > 0 ) {
30637|         AverageWaitTime.QuadPart =
30638|             Counters->TotalWaitTime.QuadPart /
30639|             | Counters->Contentions.QuadPart;
30640|     }
30641|
30642|     Debug(DEBUG_INFO,("Semaphore: %08x
30643|         | '%s'\n",Ro->Object,Ro->Name));
30644|     Debug(DEBUG_INFO,("         Total=%l64d,
30645|         | Cont=%l64d, WaitT=%l64d, MaxW=%l64d\n",
30646|         Counters->Total,
30647|         Counters->Contentions,

```



```

30642|     Counters->TotalWaitTime,
30643|     Counters->MaxWaitTime));
30644|     Debug(DEBUG_INFO,("
    | AverageW=%I64d\n",AverageWaitTime));
30645|
30646| }
30647|
30648|
30649| void pmDump_RwLockCounters (
30650|     pRegisteredObject Ro)
30651| {
30652|     ULARGE_INTEGER Average = {0};
30653|     ULARGE_INTEGER AverageR = {0};
30654|     ULARGE_INTEGER AverageW = {0};
30655|     tReaderWriterCounters
    | *Counters=&Ro->Counter.RwCounter;
30656|
30657|     Debug(DEBUG_INFO,("RwLock  : %08x
    | '%s'\n",Ro->Object,Ro->Name));
30658|     Debug(DEBUG_INFO,("          RTotal=%I64d,
    | RCont=%I64d, RTotalW=%I64d, RMaxWaitTime=%I64d,
    | MaxR=%I64d\n",
30659|         Counters->TotalReaders,
30660|         Counters->ReaderContentions,
30661|         Counters->TotalReadWaitTime,
30662|         Counters->MaxReadWaitTime,
30663|         Counters->MaxReaders) );
30664|
30665|     Debug(DEBUG_INFO,("          WTotal=%I64d,
    | WCont=%I64d, WTotalW=%I64d, WMaxWaitTime=%I64d\n",
30666|         Counters->TotalWriters,
30667|         Counters->WriterContentions,
30668|         Counters->TotalWriteWaitTime,
30669|         Counters->MaxWriteWaitTime) );
30670|
30671|     if ( Counters->ReaderContentions.QuadPart > 0 ) {
30672|         AverageR.QuadPart =
30673|             Counters->TotalReadWaitTime.QuadPart /
    | Counters->ReaderContentions.QuadPart;
30674|     }
30675|
30676|     if ( Counters->WriterContentions.QuadPart > 0 ) {
30677|         AverageW.QuadPart =
30678|             Counters->TotalWriteWaitTime.QuadPart /
    | Counters->WriterContentions.QuadPart;
30679|     }
30680|
30681|     if ( Counters->TotalReaders.QuadPart > 0 ) {
30682|         Average.QuadPart =
30683|             Counters->TotalActiveReaders.QuadPart /

```

```

    | Counters->TotalReaders.QuadPart;
30684| }
30685|
30686|   Debug(DEBUG_INFO,("      AvgRWaitTime=%!64d,
    | AvgWWaitTime=%!64d, AvgActiveReaders=%!64d\n",
30687|     AverageR,AverageW,Average) );
30688| }
30689|
30690|
30691| void pmDump_SpinlockCounters (
30692|   pRegisteredObject Ro)
30693| {
30694|   ULARGE_INTEGER AverageWaitTime = {0};
30695|   tSpinLockCounters
    | *Counters=&Ro->Counter.SpinLockCounter;
30696|
30697|   if ( Counters->Contentions.QuadPart > 0 ) {
30698|     AverageWaitTime.QuadPart =
30699|       Counters->TotalWaitTime.QuadPart /
    | Counters->Contentions.QuadPart;
30700|   }
30701|
30702|   Debug(DEBUG_INFO,("SpinLock : %08x
    | '%s'\n",Ro->Object,Ro->Name));
30703|   Debug(DEBUG_INFO,("      Total=%!64d,
    | Cont=%!64d, WaitT=%!64d, MaxW=%!64d\n",
30704|     Counters->Total,
30705|     Counters->Contentions,
30706|     Counters->TotalWaitTime,
30707|     Counters->MaxWaitTime));
30708|   Debug(DEBUG_INFO,("
    | AverageW=%!64d\n",AverageWaitTime));
30709| }
30710|
30711|
30712| void pmDumpStatistics (void)
30713| {
30714|   PLIST_ENTRY ListEntry;
30715|   KIRQL OldIrql = 0;
30716|
30717|   __try {
30718|     KeAcquireSpinLock ( &ContentionSpinLock,
    | &OldIrql );
30719|     __try {
30720|
30721|       ListEntry = RegisteredObjects.Flink;
30722|
30723|       while(ListEntry!=&RegisteredObjects) {
30724|         pRegisteredObject ro =
    | CONTAINING_RECORD(ListEntry,tRegisteredObject,ListEntry)

```

```

| ;
30725|
30726|         switch(ro->ObjectType) {
30727|             case pmSpinLock :
30728|                 pmDump_SpinlockCounters( ro );
30729|                 break;
30730|             case pmMutex    :
30731|                 pmDump_MutexCounters ( ro );
30732|                 break;
30733|             case pmSemaphore :
30734|                 pmDump_SemaphoreCounters ( ro
| );
30735|                 break;
30736|             case pmRwLock   :
30737|                 pmDump_RwLockCounters ( ro );
30738|                 break;
30739|             default:
30740|                 Debug(DEBUG_INFO,("Unknown object
| type %08x for '%s'\n",ro->ObjectType,ro->Name));
30741|                 DbgBreakPoint();
30742|             }
30743|
30744|             ListEntry = ListEntry->Flink;
30745|         }
30746|     } __finally {
30747|         KeReleaseSpinLock ( &ContentionSpinLock,
| OldIrql );
30748|     }
30749| } __except(EXCEPTION_EXECUTE_HANDLER) {
30750|     Debug(DEBUG_INFO,("Exception occurred\n"));
30751| }
30752| }
30753|
30754|
30755| #endif
30756|
30757|
30758|
30759| File Listing: CONT.h
30760|
30761| #if TRACK_CONTENTIONS
30762|
30763| /*
30764|    cont.h
30765|
30766|    Debug code for collecting contention statistics.
30767| */
30768|
30769|
30770| typedef struct sHistogram {

```

```

30771| HANDLE      WaiterThreadId;
30772| ULARGE_INTEGER Count;
30773| } tHistogram, *pHistogram;
30774|
30775|
30776| typedef struct sMutexCounters {
30777|     ULARGE_INTEGER Total;
30778|     ULARGE_INTEGER Contentions;
30779|     ULARGE_INTEGER TotalWaitTime;
30780|     ULARGE_INTEGER MaxWaitTime;
30781|     ULONG          NumHistograms;
30782|     tHistogram     Histogram[256];
30783| } tMutexCounters;
30784|
30785| typedef struct sReaderWriterCounters {
30786|     ULARGE_INTEGER TotalReaders;
30787|     ULARGE_INTEGER TotalWriters;
30788|     ULARGE_INTEGER TotalActiveReaders;
30789|     ULARGE_INTEGER MaxReaders;
30790|     ULARGE_INTEGER ReaderContentions;
30791|     ULARGE_INTEGER WriterContentions;
30792|     ULARGE_INTEGER TotalReadWaitTime;
30793|     ULARGE_INTEGER MaxReadWaitTime;
30794|     ULARGE_INTEGER TotalWriteWaitTime;
30795|     ULARGE_INTEGER MaxWriteWaitTime;
30796|     ULONG          NumHistograms;
30797|     tHistogram     Histogram[256];
30798| } tReaderWriterCounters;
30799|
30800|
30801| typedef struct sSemaphoreCounters {
30802|     ULARGE_INTEGER Total;
30803|     ULARGE_INTEGER Contentions;
30804|     ULARGE_INTEGER TotalActive;
30805|     ULONG          NumActiveMeasurement;
30806|     ULARGE_INTEGER TotalWaitTime;
30807|     ULARGE_INTEGER MaxWaitTime;
30808|     ULONG          NumHistograms;
30809|     tHistogram     Histogram [256];
30810| } tSemaphoreCounters;
30811|
30812|
30813| typedef struct sSpinLockCounters {
30814|     ULARGE_INTEGER Total;
30815|     ULARGE_INTEGER Contentions;
30816|     ULARGE_INTEGER TotalWaitTime;
30817|     ULARGE_INTEGER MaxWaitTime;
30818|     ULONG          NumHistograms;
30819|     tHistogram     Histogram[256];
30820| } tSpinLockCounters;

```

```

30821|
30822| typedef struct spmContentionCounters {
30823|     union {
30824|         tMutexCounters    MutexCounter;
30825|         tReaderWriterCounters RwCounter;
30826|         tSemaphoreCounters SemaphoreCounter;
30827|         tSpinLockCounters SpinLockCounter;
30828|     };
30829| } tpmContentionCounters,*ppmContentionCounters;
30830|
30831|
30832| extern KSPIN_LOCK ContentionSpinLock;
30833|
30834|
30835| // undefine these so the functions get called instead.
30836| #undef pmAcquireSemaphore
30837| #undef pmAcquireMutex
30838| #undef pmAcquireSpinLock
30839| #undef pmAcquireReaderLock
30840| #undef pmAcquireWriterLock
30841| #undef pmRegisterObject
30842| #undef pmDeRegisterObject
30843| #undef pmWaitForMultipleObjects
30844|
30845| NTSTATUS pmWaitForMultipleObjects( PVOID Objects[],
    | ULONG Num, PLARGE_INTEGER Timeout );
30846|
30847| NTSTATUS pmRegisterObject( PVOID Object, char *Name,
    | tPmateObjectType ObjectType );
30848| NTSTATUS pmDeRegisterObject( PVOID Object );
30849|
30850| NTSTATUS pmAcquireSemaphore (
30851|     PKSEMAPHORE Semaphore,
30852|     PLARGE_INTEGER Timeout );
30853|
30854| void pmAcquireMutex (
30855|     PFAST_MUTEX fastMutex,
30856|     PLARGE_INTEGER Timeout );
30857|
30858| void pmAcquireSpinLock (
30859|     PKSPIN_LOCK spinLock,
30860|     PKIRQL callersOldIrql );
30861|
30862| BOOLEAN pmAcquireReaderLock (
30863|     PERESOURCE Resource,
30864|     BOOLEAN Wait);
30865|
30866| BOOLEAN pmAcquireWriterLock (
30867|     PERESOURCE Resource,
30868|     BOOLEAN Wait);

```

```

30869|
30870| void pmDumpStatistics (void);
30871|
30872| #endif // TRACK_CONTENTIONS
30873|
30874|
30875|
30876| File Listing: CREATE.cpp
30877|
30878| #include "precomp.h"
30879|
30880|
30881| /*-----
    | -----*/
30882| NTSTATUS
30883| PSMANCreate(
30884|     IN PDEVICE_OBJECT DeviceObject,
30885|     IN PIRP Irp
30886| )
30887|
30888| /*++
30889|
30890| Routine Description:
30891|
30892|     Passes the device object to the correct handler
30893|
30894| Arguments:
30895|
30896|     DeviceObject - Context for the activity.
30897|     Irp          - The device control argument block.
30898|
30899| Return Value:
30900|
30901|     NT Status
30902|
30903| --*/
30904|
30905| {
30906| #ifdef DEBUG
30907|     // Code for hunting down weird bug where return
    | address appears to be corrupted... (part 1 of 2)
30908|     PVOID Daddy1 = NULL;
30909|     PVOID Grandpa1 = NULL;
30910|     RtlGetCallersAddress(&Daddy1, &Grandpa1);
30911| #endif /*DEBUG*/
30912|     ULONG FlowTrace = 0; // also for tracking down
    | weird bug
30913|
30914|     NTSTATUS Status = STATUS_UNSUCCESSFUL;
30915|

```

```

30916| FlowTrace ^= 0x20;
30917| switch(PsmGetObjectTypes(DeviceObject)) {
30918|     case OBJECT_INTERNAL :
30919|         FlowTrace ^= 0x01;
30920|         Status = PSMAN_CREATE_OBJECT(DeviceObject,
30921| | Irp);
30922|         FlowTrace ^= 0x01;
30923|         break;
30924|     case OBJECT_FILTEREDDISK :
30925|         FlowTrace ^= 0x02;
30926|         Status = PSMAN_CREATE_DEVICE(DeviceObject,
30927| | Irp);
30928|         FlowTrace ^= 0x02;
30929|         break;
30930|     case OBJECT_VIRTUALDISK :
30931|         FlowTrace ^= 0x04;
30932|         Status = PSMAN_CREATE_VDISK(DeviceObject,
30933| | Irp);
30934|         FlowTrace ^= 0x04;
30935|         break;
30936|     case OBJECT_FS_FILTER :
30937|         FlowTrace ^= 0x08;
30938|         Status = PSMAN_CREATE_FS_FILTER(DeviceObject,
30939| | Irp);
30940|         FlowTrace ^= 0x08;
30941|         break;
30942|     case OBJECT_FS_OBJECT :
30943|         FlowTrace ^= 0x10;
30944|         Status = PSMAN_CREATE_FS_OBJECT(DeviceObject,
30945| | Irp);
30946|         FlowTrace ^= 0x10;
30947|         break;
30948|     default:
30949|         Irp->IoStatus.Status = STATUS_NO_SUCH_DEVICE;
30950|         Irp->IoStatus.Information = 0 ;
30951|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
30952|         break;
30953| }
30954| FlowTrace ^= 0x20;
30955| ASSERT(FlowTrace==0);
30956| #ifdef DEBUG
30957| // Code for hunting down weird bug where return
30958| // address appears to be corrupted... (part 2 of 2)
30959| PVOID Daddy2 = NULL;
30960| PVOID Grandpa2 = NULL;
30961| RtlGetCallersAddress (&Daddy2, &Grandpa2);
30962| ASSERT(Daddy1 == Daddy2);
30963| ASSERT(Grandpa1 == Grandpa2);

```

```

30959| #endif /*DEBUG*/
30960|
30961|     return Status;
30962|
30963| } // end PSMCreate()
30964|
30965| pOT_USER InternalCreateUser( PEPROCESS ProcessId,
    | PETHREAD ThreadId, PFILE_OBJECT FileObject )
30966| {
30967|     pOT_USER User = (pOT_USER) MemAllocatePoolWithTag
    | (PagedPool, sizeof(tOT_USER),USERTAG);
30968|     if(User) {
30969|         RtlZeroMemory(User,sizeof(tOT_USER));
30970|         User->ProcessID = ProcessId;
30971|         User->ThreadID = ThreadId;
30972|         User->Persistent = FALSE;
30973|         User->SaveTempOnExit = FALSE;
30974|         User->Open = 0;
30975|         User->NumOpenSnapShots = 0;
30976|         User->FileObject = FileObject;
30977|         User->ErrorEvent= NULL;
30978|         User->AbortEvent= NULL;
30979|         InitializeListHead(&User->SnapShots);
30980|         AddPSMUser(User);
30981|     }
30982|     return User;
30983| }
30984|
30985|
30986| /*-----
    | -----*/
30987| STATIC NTSTATUS
30988| PSMCreateObject(
30989|     IN PDEVICE_OBJECT DeviceObject,
30990|     IN PIRP Irp
30991| )
30992|
30993| /*++
30994|
30995| Routine Description:
30996|
30997|     This routine services open commands. It establishes
30998|     the driver's existence by returning status success.
30999|
31000| Arguments:
31001|
31002|     DeviceObject - Context for the activity.
31003|     Irp          - The device control argument block.
31004|
31005| Return Value:

```



```

31006|
31007|    NT Status
31008|
31009| --*/
31010|
31011| {
31012| #ifdef DEBUG
31013|    // Code for hunting down weird bug where return
    | address appears to be corrupted... (part 1 of 2)
31014|    PVOID Daddy1 = NULL;
31015|    PVOID Grandpa1 = NULL;
31016|    RtlGetCallersAddress(&Daddy1, &Grandpa1);
31017| #endif /*DEBUG*/
31018|
31019|    pOT_USER User=NULL;
31020|    NTSTATUS Status=STATUS_SUCCESS;
31021|
31022|    NOT_REFERENCED(DeviceObject);
31023|    PAGED_CODE();
31024|
31025|    /*lint -save -e746 */
31026|    Debug(DEBUG_PROCCALL,("PSManCreateObject Called:
    | PsGetCurrentProcess=%08x "
31027|        "PsGetCurrentThread=%08x "
31028|        "IoGetCurrentProcess=%08x "
31029|        "KeGetCurrentThread=%08x "
31030|        "User Thread=%08x "
31031|        "OFO=%08x\n",
31032|        PsGetCurrentProcess(),
31033|        PsGetCurrentThread(),
31034|        IoGetCurrentProcess(),
31035|        KeGetCurrentThread(),
31036|        Irp->Tail.Overlay.Thread,
31037|
    | Irp->Tail.Overlay.OriginalFileObject
31038|        ));
31039|    /*lint -restore */
31040|
31041|    // keep track of who has us open.
31042|    User =
    | InternalCreateUser(PsGetCurrentProcess(),PsGetCurrentThr
    | ead(),Irp->Tail.Overlay.OriginalFileObject);
31043|
31044|    if(User) {
31045|        Status = Irp->IoStatus.Status = STATUS_SUCCESS;
31046|    } else {
31047|        Status = Irp->IoStatus.Status =
    | STATUS_INSUFFICIENT_RESOURCES;
31048|    }
31049|    Irp->IoStatus.Information = 0;

```

```

31050|
31051| IoCompleteRequest(Irp, IO_NO_INCREMENT);
31052| Debug(DEBUG_PROCCALL,("PSManCreateObject Done\n"));
31053|
31054| #ifdef DEBUG
31055| // Code for hunting down weird bug where return
    | address appears to be corrupted... (part 2 of 2)
31056| PVOID Daddy2 = NULL;
31057| PVOID Grandpa2 = NULL;
31058| RtlGetCallersAddress (&Daddy2, &Grandpa2);
31059| ASSERT(Daddy1 == Daddy2);
31060| ASSERT(Grandpa1 == Grandpa2);
31061| #endif /*DEBUG*/
31062|
31063| return Status;
31064|
31065| } // end PSManCreateObject()
31066|
31067|
31068| /*-----
    | -----*/
31069| STATIC NTSTATUS
31070| PSManCreateDevice(
31071|     IN PDEVICE_OBJECT DeviceObject,
31072|     IN PIRP Irp
31073| )
31074|
31075| /*++
31076|
31077| Routine Description:
31078|
31079|     This routine services open commands. It establishes
31080|     the driver's existence by returning status success.
31081|
31082| Arguments:
31083|
31084|     DeviceObject - Context for the activity.
31085|     Irp          - The device control argument block.
31086|
31087| Return Value:
31088|
31089|     NT Status
31090|
31091| --*/
31092|
31093| {
31094| #ifdef DEBUG
31095| // Code for hunting down weird bug where return
    | address appears to be corrupted... (part 1 of 2)
31096| PVOID Daddy1 = NULL;

```

```

31097|   PVOID Grandpa1 = NULL;
31098|   RtlGetCallersAddress(&Daddy1, &Grandpa1);
31099| #endif /*DEBUG*/
31100|
31101|   NTSTATUS Status;
31102|   PIO_STACK_LOCATION currentIrpStack =
        | IoGetCurrentIrpStackLocation(Irp);
31103|   TRACE( TRACE_CREATE,
31104|
        | currentIrpStack->Parameters.Read.ByteOffset.HighPart,
31105|
        | currentIrpStack->Parameters.Read.ByteOffset.LowPart,
31106|         currentIrpStack->Parameters.Read.Length,
31107|         currentIrpStack->Parameters.Read.Key,
31108|         "");
31109|
31110| #ifdef DEBUG
31111|   if(PsmActive) {
31112|       Debug(DEBUG_CREATE |
        | DEBUG_PROCCALL,("PSManCreateDevice Called\n"));
31113|   }
31114| #endif
31115|   Status = PSManPassThru( DeviceObject, Irp );
31116| #ifdef DEBUG
31117|   if(PsmActive) {
31118|       Debug(DEBUG_CREATE |
        | DEBUG_PROCCALL,("PSManCreateDevice Done\n"));
31119|   }
31120| #endif
31121|
31122| #ifdef DEBUG
31123|   // Code for hunting down weird bug where return
        | address appears to be corrupted... (part 2 of 2)
31124|   PVOID Daddy2 = NULL;
31125|   PVOID Grandpa2 = NULL;
31126|   RtlGetCallersAddress (&Daddy2, &Grandpa2);
31127|   ASSERT(Daddy1 == Daddy2);
31128|   ASSERT(Grandpa1 == Grandpa2);
31129| #endif /*DEBUG*/
31130|   return Status;
31131|
31132| } // end PSManCreateDevice()
31133|
31134| /*-----*/
        | -----*/
31135| STATIC NTSTATUS PSManCreateVDisk(
31136|   IN PDEVICE_OBJECT DeviceObject,
31137|   IN PIRP Irp
31138|   )
31139| {

```

```

31140| #ifdef DEBUG
31141|    // Code for hunting down weird bug where return
    | address appears to be corrupted... (part 1 of 2)
31142|    PVOID Daddy1 = NULL;
31143|    PVOID Grandpa1 = NULL;
31144|    RtlGetCallersAddress(&Daddy1, &Grandpa1);
31145| #endif /*DEBUG*/
31146|
31147|    NTSTATUS Status=STATUS_SUCCESS;
31148|
31149|    Debug(DEBUG_PROCCALL |
    | DEBUG_CREATE,("PsmanCreateVDisk Called Dev=%p,
    | Irp=%p\n",DeviceObject,Irp));
31150|    Irp->IoStatus.Information = 0;
31151|    Irp->IoStatus.Status = Status;
31152|    IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
31153|    Debug(DEBUG_PROCCALL |
    | DEBUG_CREATE,("PsmanCreateVDisk Done\n"));
31154|
31155| #ifdef DEBUG
31156|    // Code for hunting down weird bug where return
    | address appears to be corrupted... (part 2 of 2)
31157|    PVOID Daddy2 = NULL;
31158|    PVOID Grandpa2 = NULL;
31159|    RtlGetCallersAddress (&Daddy2, &Grandpa2);
31160|    ASSERT(Daddy1 == Daddy2);
31161|    ASSERT(Grandpa1 == Grandpa2);
31162| #endif /*DEBUG*/
31163|
31164|    return Status;
31165| }
31166|
31167|
31168| /*-----
    | -----*/
31169| STATIC NTSTATUS
31170| PSManCreateFSFilter(
31171|     IN PDEVICE_OBJECT DeviceObject,
31172|     IN PIRP Irp
31173| )
31174|
31175| /*++
31176|
31177| Routine Description:
31178|
31179|     This routine services open commands. It establishes
31180|     the driver's existence by returning status success.
31181|
31182| Arguments:
31183|

```

```

31184| DeviceObject - Context for the activity.
31185| Irp      - The device control argument block.
31186|
31187| Return Value:
31188|
31189| NT Status
31190|
31191| --*/
31192|
31193| {
31194| #ifdef DEBUG
31195|     // Code for hunting down weird bug where return
        | address appears to be corrupted... (part 1 of 2)
31196|     PVOID Daddy1 = NULL;
31197|     PVOID Grandpa1 = NULL;
31198|     RtlGetCallersAddress(&Daddy1, &Grandpa1);
31199| #endif /*DEBUG*/
31200|
31201|     NTSTATUS Status = SfCreate(DeviceObject,Irp);
31202|
31203| #ifdef DEBUG
31204|     // Code for hunting down weird bug where return
        | address appears to be corrupted... (part 2 of 2)
31205|     PVOID Daddy2 = NULL;
31206|     PVOID Grandpa2 = NULL;
31207|     RtlGetCallersAddress (&Daddy2, &Grandpa2);
31208|     ASSERT(Daddy1 == Daddy2);
31209|     ASSERT(Grandpa1 == Grandpa2);
31210| #endif /*DEBUG*/
31211|
31212|     return Status;
31213| } // end PSMANCreateFilter()
31214|
31215| /*-----
        | -----*/
31216| STATIC NTSTATUS PSMANCreateFSObject(
31217|     IN PDEVICE_OBJECT DeviceObject,
31218|     IN PIRP Irp
31219| )
31220| {
31221| #ifdef DEBUG
31222|     // Code for hunting down weird bug where return
        | address appears to be corrupted... (part 1 of 2)
31223|     PVOID Daddy1 = NULL;
31224|     PVOID Grandpa1 = NULL;
31225|     RtlGetCallersAddress(&Daddy1, &Grandpa1);
31226| #endif /*DEBUG*/
31227|
31228|     NTSTATUS Status=STATUS_SUCCESS;
31229|

```

```

31230|  Debug(DEBUG_PROCCALL |
      | DEBUG_CREATE,("PsmanCreateFSObject Called Dev=%p,
      | Irp=%p\n",DeviceObject,Irp));
31231|  Irp->IoStatus.Information = FILE_OPENED;
31232|  Irp->IoStatus.Status = Status;
31233|  IoCompleteRequest (Irp, IO_DISK_INCREMENT) ;
31234|  Debug(DEBUG_PROCCALL |
      | DEBUG_CREATE,("PsmanCreateFSObject Done\n"));
31235|
31236| #ifdef DEBUG
31237|  // Code for hunting down weird bug where return
      | address appears to be corrupted... (part 2 of 2)
31238|  PVOID Daddy2 = NULL;
31239|  PVOID Grandpa2 = NULL;
31240|  RtlGetCallersAddress (&Daddy2, &Grandpa2);
31241|  ASSERT(Daddy1 == Daddy2);
31242|  ASSERT(Grandpa1 == Grandpa2);
31243| #endif /*DEBUG*/
31244|
31245|  return Status;
31246| }
31247|
31248|
31249|
31250| File Listing: CREATE.h
31251|
31252| NTSTATUS
31253| PSMANCreate(
31254|     IN PDEVICE_OBJECT DeviceObject,
31255|     IN PIRP Irp
31256| );
31257|
31258| STATIC NTSTATUS
31259| PSMANCreateObject(
31260|     IN PDEVICE_OBJECT DeviceObject,
31261|     IN PIRP Irp
31262| );
31263|
31264| STATIC NTSTATUS
31265| PSMANCreateDevice(
31266|     IN PDEVICE_OBJECT DeviceObject,
31267|     IN PIRP Irp
31268| );
31269|
31270| STATIC NTSTATUS PSMANCreateVDisk(
31271|     IN PDEVICE_OBJECT DeviceObject,
31272|     IN PIRP Irp
31273| );
31274|
31275| STATIC NTSTATUS PSMANCreateFSObject(

```

```

31276| IN PDEVICE_OBJECT DeviceObject,
31277| IN PIRP Irp
31278| );
31279|
31280| STATIC NTSTATUS PSMCreateFSFilter(
31281| IN PDEVICE_OBJECT DeviceObject,
31282| IN PIRP Irp
31283| );
31284|
31285|
31286| pOT_USER InternalCreateUser( PEPROCESS ProcessId,
    | PETHREAD ThreadId, PFILE_OBJECT FileObject );
31287|
31288|
31289|
31290| File Listing: DCPSM.cpp
31291|
31292| #include "precomp.h"
31293|
31294| #ifdef ALLOC_PRAGMA_DO_NOT_DO
31295|     #pragma alloc_text(PAGE, SbPreInit)
31296|     #pragma alloc_text(PAGE, SbOpenPSM)
31297|     #pragma alloc_text(PAGE, SbClosePSM)
31298| #endif
31299|
31300| //STATIC NTSTATUS AddDeviceToList( pkSnapShotEntry
    | *List, pkSnapShotEntry SnapShot);
31301| NTSTATUS ValidateKernelSnapShotPointer ( PVOID
    | KernelPointer );
31302|
31303|
31304| /*-----*/
    | -----*/
31305| STATIC void DumpDebugPointers ( void )
31306| {
31307|     Debug(DEBUG_DCPSM,("ThreadObjects %p WriterObject
    | %p ThreadsAwake %d NumThreads %d\n",
31308|         ThreadObjects,
31309|         WriteThreadObject,
31310|         ThreadsAwake,
31311|         NumberOfThreads
31312|         ));
31313|
31314|     //LIST_ENTRY   ThreadsWorkToDoQueue={0};
31315|     //KSPIN_LOCK   ThreadsWorkToDoSpinLock={0};
31316|
31317|     //LIST_ENTRY   WriteQueue={0};
31318|     //KSPIN_LOCK   WriteSpinLock={0};
31319| }
31320|

```

```

31321|
31322|
31323| /*-----
    | -----*/
31324| /*
31325|     Given a NT device name will return our object
    | associated with it.
31326| */
31327| PDEVICE_OBJECT GetObjectFromName( WCHAR *Name )
31328| {
31329|     PDEVICE_OBJECT DevObj ;
31330|     __try {
31331|         DevObj = PManDriverObject->DeviceObject;
31332|
31333|         while ( DevObj ) {
31334|             // if a filtered disk
31335|             if (
    | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
31336|                 PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DevObj);
31337|                 if ( _wcsicmp(DevExt->Name,Name)==0 ) {
31338|                     break;
31339|                 }
31340|             } // if filtered disk
31341|
31342|             DevObj = DevObj->NextDevice;
31343|         } // while(DevObj)
31344|     } __except
    | (ExceptionFilter(GetExceptionInformation())) {
31345|         Debug(DEBUG_DCPDM,("GetObjectForName: Exception
    | %08x",GetExceptionCode()));
31346|         DevObj = NULL; // indicate error to caller -
    | don't want to return wrong DevObj
31347|     }
31348|     return DevObj;
31349| }
31350|
31351| PDEVICE_OBJECT GetVdiskObjectForName( WCHAR *Name,
    | ULONG Instance )
31352| {
31353|     __try {
31354|         PVDISK_EXTENSION DevExt;
31355|         PFILTERED_EXTENSION FiltExt;
31356|         PDEVICE_OBJECT DevObj =
    | PManDriverObject->DeviceObject;
31357|         while ( DevObj ) {
31358|
31359|             DevExt =
    | (PVDISK_EXTENSION)GetDeviceExtension(DevObj);
31360|             if (

```



```

    | (PsmGetObjectType(DevObj)==OBJECT_VIRTUALDISK) &&
31361|         (DevExt->Instance == Instance ) ) {
31362|         if ( DevExt->PSMDevice ) {
31363|             FiltExt =
    | GetFilteredExtension(DevExt->PSMDevice);
31364|             if (
    | _wcsicmp(Name,FiltExt->Name)==0 ) {
31365|                 return DevObj;
31366|             }
31367|         }
31368|     }
31369|
31370|     DevObj = DevObj->NextDevice;
31371| }
31372| } __except
    | (ExceptionFilter(GetExceptionInformation())) {
31373|     Debug(DEBUG_DCPSM,("GetVdiskObjectForName:
    | Exception %08x",GetExceptionCode()));
31374| }
31375|     Debug(DEBUG_DCPSM,("GetVdiskObjectForName: Name
    | '%S' not found\n",Name));
31376|     return NULL;
31377| }
31378|
31379|
31380|
31381| /*
31382|     Given a NT device name will return our object
    | associated with it.
31383| */
31384| PDEVICE_OBJECT GetObjectFromVDiskName( WCHAR *Name )
31385| {
31386|     PDEVICE_OBJECT DevObj ;
31387|     __try {
31388|         DevObj = PSMAN_DRIVER_OBJECT->DeviceObject;
31389|         WCHAR VD[256]={0};
31390|
31391|         while ( DevObj ) {
31392|             // if a virtual disk
31393|             if (
    | PsmGetObjectType(DevObj)==OBJECT_VIRTUALDISK ) {
31394|                 PVDISK_EXTENSION DevExt =
    | GetVDiskExtension(DevObj);
31395|
31396|                 | swprintf(VD,L"\\Device\\PsmDevices_%04x\\%s_%d",PSM_LOW_
    | COMPATIBLE_VERSION,DevExt->Name,DevExt->Instance);
31397|
31398|                 if ( _wcsicmp(VD,Name)==0 ) {
31399|                     break;

```

```

31400|         }
31401|     } // if virtual disk
31402|
31403|         DevObj = DevObj->NextDevice;
31404|     } // while(DevObj)
31405| } __except
    | (ExceptionFilter(GetExceptionInformation())) {
31406|     Debug(DEBUG_DCPSM,("GetObjectFromVDiskName:
    | Exception %08x",GetExceptionCode()));
31407| }
31408|
31409| return DevObj;
31410| }
31411|
31412| /*-----
    | -----*/
31413| /*
31414|     Given a Win32 name will return our object
    | associated with it.
31415|     A win 32 name begins with \DosDevices\ or \??\
31416|     ie: \DosDevices\C:\dir\filename.ext
31417| */
31418| PDEVICE_OBJECT GetObjectFromWin32Name( WCHAR *FileName
    | )
31419| {
31420|     UNICODE_STRING  UniName={0};
31421|     OBJECT_ATTRIBUTES ObjectAttributes={0};
31422|     NTSTATUS        Status=0;
31423|     PDEVICE_OBJECT  DevObj=NULL;
31424|     WCHAR           Name[256]={0};
31425|     WCHAR           *p=NULL;
31426|     HANDLE          SymHandle=NULL;
31427|     ULONG           Ret=0;
31428|
31429|     // convert from \DosDevices\C:\dir\filename.ext
31430|     //      to \DosDevices\C:
31431|     // find second slash \c:\dir\filename.ext
31432|     p = wcschr(FileName+1,L'\\');
31433|     if ( !p ) {
31434|         Debug(DEBUG_DCPSM,("GetObjectFromWin32Name: Not
    | valid name '%S'\n",FileName));
31435|         return NULL;
31436|     }
31437|     // find third slash \dir\filename.ext
31438|     p = wcschr(p+1,L'\\');
31439|     if ( !p ) {
31440|         Debug(DEBUG_DCPSM,("GetObjectFromWin32Name: Not
    | valid name '%S'\n",FileName));
31441|         return NULL;
31442|     }

```

```

31443|
31444|  ASSERT(p-FileName<256);
31445|
31446|  // copy and get rid of tail
31447|  wcsncpy(Name,FileName,p-FileName);
31448|  Name[p-FileName]=0;
31449|
31450|  Debug(DEBUG_DCPSM,("GetObjectFromWin32Name: Looking
    | for '%S'\n",Name));
31451|  RtlInitUnicodeString( &UniName, Name);
31452|
31453|  InitializeObjectAttributes ( &ObjectAttributes,
31454|                               &UniName,
31455|                               OBJ_CASE_INSENSITIVE,
31456|                               NULL,
31457|                               NULL );
31458|
31459|  Status = ZwOpenSymbolicLinkObject( &SymHandle,
    | STANDARD_RIGHTS_READ, &ObjectAttributes);
31460|  if ( NT_SUCCESS(Status) ) {
31461|
31462|      UniName.Length = 0;
31463|      UniName.MaximumLength = 256;
31464|      Status =
    | ZwQuerySymbolicLinkObject(SymHandle,&UniName,&Ret);
31465|      if ( NT_SUCCESS(Status) ) {
31466|
31467|          Debug(DEBUG_DCPSM,("GetObjectFromWin32Name:
    | Device Name = '%wZ'\n",&UniName));
31468|          DevObj = GetObjectFromName(UniName.Buffer);
31469|          if ( !DevObj ) {
31470|
    | Debug(DEBUG_DCPSM,("GetObjectFromWin32Name: Unable to
    | find device object\n"));
31471|          }
31472|
31473|      } else {
31474|          Debug(DEBUG_DCPSM,("GetObjectFromWin32Name:
    | Error %08x opening symlink '%S'\n",Status,Name));
31475|      }
31476|
31477|      ZwClose(SymHandle);
31478|      SymHandle = NULL;
31479|  } else {
31480|      Debug(DEBUG_DCPSM,("Error %08x\n",Status));
31481|  }
31482|
31483|
31484| #if 0
31485| // win2k specific way, but the nt 4 way should work

```

```

| also
31486| // so lets not do this unless we need to.
31487|   InitializeObjectAttributes ( &ObjectAttributes,
31488|                               &UniName,
31489|                               OBJ_CASE_INSENSITIVE,
31490|                               NULL,
31491|                               NULL );
31492|
31493|   Status = ZwCreateFile( &FileHandle,
31494|                           0,
31495|                           | // desired access
31496|                           &ObjectAttributes,
31497|                           | // object attributes
31498|                           &IoStatus,
31499|                           NULL,
31500|                           | // alloc size
31501|                           FILE_ATTRIBUTE_NORMAL,
31502|                           | // file attributes
31503|                           FILE_SHARE_WRITE |
31504|                           | FILE_SHARE_READ,
31505|                           | // share access
31506|                           FILE_OPEN,
31507|                           | // create disposition
31508|                           FILE_SYNCHRONOUS_IO_NONALERT,
31509|                           | // create options
31510|                           NULL, // eabuffer
31511|                           0 ); // ealength
31512|   if ( NT_SUCCESS(Status) ) {
31513|       ULONG CurrentSize = 100;
31514|       PMOUNTDEV_NAME MN;
31515|       do {
31516|           MN =
31517|           | MemAllocatePoolWithTag(PagedPool,CurrentSize+sizeof(MOUN
31518|           | TDEV_NAME),TEMPTAG);
31519|           if ( MN ) {
31520|               Status = ZwDeviceIoControlFile(
31521|               | FileHandle,NULL,NULL,NULL,&IoStatusBlock,
31522|               | IOCTL_MOUNTDEV_QUERY_DEVICE_NAME,
31523|               | NULL,0,MN,CurrentSize+sizeof(MOUNTDEV_NAME)
31524|               );
31525|               if ( Status==STATUS_BUFFER_TOO_SMALL )
31526|               | {
31527|                   FREE_POINTER(MN);
31528|                   CurrentSize*=2;

```

```

31520|         }
31521|     } else {
31522|         Status = STATUS_INSUFFICIENT_RESOURCES;
31523|     }
31524| } while ( Status==STATUS_BUFFER_TOO_SMALL );
31525|
31526| if ( NT_SUCCESS(Status) ) {
31527|     Debug(DEBUG_DCPSM,("GetObjectFromWin32Name:
| '%s' = '%s'\n",Name,MN->Name));
31528|     DevObj = GetObjectFromName(MN->Name);
31529|     if ( !DevObj ) {
31530|
| Debug(DEBUG_DCPSM,("GetObjectFromWin32Name: Unable to
| find device object\n"));
31531|     }
31532| } else {
31533|     Debug(DEBUG_DCPSM,("GetObjectFromWin32Name:
| Error %08x sending ioctl to '%s'\n",Status,Name));
31534| }
31535|
31536| if ( MN ) {
31537|     FREE_POINTER(MN);
31538| }
31539| ZwClose(FileHandle);
31540| } else {
31541|     Debug(DEBUG_DCPSM,("GetObjectFromWin32Name:
| Error %08x opening '%s'\n",Status,Name));
31542| }
31543| #endif
31544| return DevObj;
31545| }
31546|
31547| /*-----
| -----*/
31548| /*
31549|     will return OUR object associated with the
| given drive letter
31550|
31551| */
31552| PDEVICE_OBJECT GetObjectFromDriveLetter( WCHAR
| DriveLetter )
31553| {
31554|     WCHAR        DriveName[20] =
| L"\\DosDevices\\C:\\";
31555|
31556|     DriveName[12] = DriveLetter;
31557|
31558|     return GetObjectFromWin32Name(DriveName);
31559| }
31560|

```

```

31561| /*-----
| -----*/
31562| /*
31563|  Add non physical devices to list of devices to psm.
31564| */
31565| STATIC NTSTATUS FindAndAddVolumesForDevice(
| PDEVICE_OBJECT PhysicalDevice, PVOID *ObjectTable,
| ULONG *NumObjects )
31566| {
31567| #if _WIN32_WINNT < 0x0500
31568|  PDEVICE_OBJECT DevObj =
| PSMAN_DRIVER_OBJECT->DeviceObject;
31569|  NTSTATUS Status=STATUS_SUCCESS;
31570|
31571|  while ( DevObj ) {
31572|      // if a filtered disk
31573|      if (
| PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
31574|          PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(DevObj);
31575|
31576|          // if this volume is on this device, but is
| not the physical device itself
31577|          if (
| (DevExt->PhysicalDevice==PhysicalDevice) &&
| (!DevExt->IsPhysical) ) {
31578|
31579|              // okay we now have a logical volume.
31580|              ObjectTable[*NumObjects] =
| &(DevExt->WriteEvent);
31581|              (*NumObjects)++;
31582|
31583|              // say we want event notification.
31584|
| InterlockedIncrement((PLONG)&DevExt->SignalWrite);
31585|          }
31586|      } // if filtered disk
31587|
31588|      DevObj = DevObj->NextDevice;
31589|  } // while(DevObj)
31590|  return Status;
31591| #else
31592|  return STATUS_NOT_IMPLEMENTED;
31593| #endif
31594|
31595| }
31596|
31597| /*-----
| -----*/
31598| STATIC NTSTATUS MakeVolumeListToPsm(

```

```

    | pOpenTransactionInInternal In, PVOID *ObjectTable,
    | ULONG *NumObjects )
31599| {
31600|     ULONG i;
31601|     PDEVICE_OBJECT DevObj=NULL;
31602|     PFILTERED_EXTENSION DevExt=NULL;
31603|     NTSTATUS Status=STATUS_SUCCESS;
31604|
31605|     Debug(DEBUG_PROCCALL,("MakeVolumeListToPsm\n"));
31606|     for ( i=0;i<In->NumberOfDevices;i++ ) {
31607|         DevObj = (PDEVICE_OBJECT) GetObjectFromName(
            | (WCHAR *)DN_MakePointer(In,In->DeviceName[i]) );
31608|         if ( DevObj ) {
31609|             DevExt = GetFilteredExtension(DevObj);
31610|
31611|             // okay we now have a device.
31612|             ObjectTable[*NumObjects] =
                | &(DevExt->WriteEvent);
31613|             (*NumObjects)++;
31614|
31615|             // say we want event notification.
31616|
            | InterlockedIncrement((PLONG)&DevExt->SignalWrite);
31617|
31618|             // enable psm on physical device also
31619|
31620|             if ( !DevExt->IsPhysical ) {
31621| #if 0
31622| // FIXFIXFIX !FIX!FIX!FIX if the physical device is
            | written to, we will
31623|             no longer psm it. this may or may not
            | be a problem, more so on nt4 than
31624|             w2k.
31625|             PFILTERED_EXTENSION PhyExt =
                | GetFilteredExtension(DevExt->PhysicalDevice);
31626|
31627|             // only if we havent done it yet.
31628|             // FIXFIXFIX if we allow multiple waits
            | at the same time fix this
31629|             if ( !PhyExt->SignalWrite ) {
31630|
31631|                 ObjectTable[*NumObjects] =
                    | &(PhyExt->WriteEvent);
31632|                 (*NumObjects)++;
31633|
31634|
            | InterlockedIncrement(&PhyExt->SignalWrite);
31635|
31636|             // make sure null, as we dont
            | allocate space on the physical side

```

```

31637|         PhyExt->PSMSectors = NULL;
31638|     }
31639| #endif
31640|     } else {
31641|         // okay, the user said to do the whole
        | drive, map in all partitions for this device
31642|         Status =
        | FindAndAddVolumesForDevice(DevObj,ObjectTable,NumObjects
        | );
31643|         if ( !INT_SUCCESS(Status) ) {
31644|             Debug(DEBUG_DCPSM,("PSMan: FAAVFD
        | failed with %08x\n",Status));
31645|             break;
31646|         }
31647|     }
31648|     } else {
31649|         Debug(DEBUG_DCPSM,("PSMan: Unable to find
        | object for device
        | '%S'\n",DN_MakePointer(In,In->DeviceName[i])));
31650|         Status = STATUS_NO_SUCH_DEVICE;
31651|     }
31652| }
31653|
31654| if ( !INT_SUCCESS(Status) ) {
31655|     // clear memory allocated and flags
31656|     DevObj = PSManDriverObject->DeviceObject;
31657|     while ( DevObj ) {
31658|         // if a filtered disk
31659|         if (
            | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
31660|             ULONG k;
31661|             DevExt = GetFilteredExtension(DevObj);
31662|
31663|             for ( k=0;k<*NumObjects;k++ ) {
31664|                 if ( &DevExt->WriteEvent ==
            | ObjectTable[k] ) {
31665|                     | InterlockedDecrement((PLONG)&DevExt->SignalWrite);
31666|                 }
31667|             }
31668|
31669|         } // if filtered disk
31670|
31671|         DevObj = DevObj->NextDevice;
31672|     } // while(DevObj)
31673| }
31674|
31675| return Status;
31676| }
31677|

```



```

31678| /*-----
| -----*/
31679| STATIC void FlushVolumeList( pOpenTransactionInternal
| In )
31680| {
31681|     ULONG i;
31682|
31683|     for ( i=0;i<In->NumberOfDevices;i++ ) {
31684|         FlushVolume((WCHAR
| *)DN_MakePointer(In,In->DeviceName[i]));
31685|     }
31686| }
31687|
31688| // Time we went into our wait for no io mode.
31689| STATIC LARGE_INTEGER CurrentTime={0};
31690| STATIC ULONG     SecondsToWait=0;
31691|
31692|
31693| // make sure the volumes are good
31694| // 1. Check for 512 byte sectors since that is all we
| support
31695| /*-----
| -----*/
31696| STATIC NTSTATUS VerifyGoodVolumeList(
| pOpenTransactionInternal In )
31697| {
31698|     ULONG i;
31699|     PDEVICE_OBJECT DevObj=NULL;
31700|     NTSTATUS Status = STATUS_SUCCESS;
31701|     PAGED_CODE();
31702|
31703|     Debug(DEBUG_PROCCALL,("VerifyGoodVolumeList\n"));
31704|     for ( i=0;i<In->NumberOfDevices;i++ ) {
31705|         DevObj = GetObjectFromName((WCHAR
| *)DN_MakePointer(In,In->DeviceName[i]));
31706|         if ( DevObj ) {
31707|             PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(DevObj);
31708|
31709|             if ( DevExt->NotActive ) {
31710|                 Debug(DEBUG_DCPSM,("Error! Device %S is
| not active!\n",DevExt->Name));
31711|                 Status = PSM_ERROR_VOLUME_NOT_ACTIVE;
31712|                 break;
31713|             }
31714|
31715|             if ( DevExt->BytesPerSector!=512 ) {
31716|                 Debug(DEBUG_DCPSM,("Error! Device %08x
| does not have a blocksize of 512!\n",DevObj));
31717|                 Status = STATUS_INVALID_PARAMETER;

```

```

31718|         break;
31719|     }
31720| } else {
31721|     // well off to a bad start, no device
31722|     Debug(DEBUG_DCPSM,("Error! Device does not
    | exist!\n"));
31723|     Status = STATUS_NO_SUCH_DEVICE;
31724|     break;
31725| }
31726| }
31727|
31728| #if _WIN32_WINNT < 0x0500
31729|     // check to see if cache file is on a good volume
31730|     if ( Status==STATUS_SUCCESS ) {
31731|         // \DosDevices\C:\temp\psm3.tmp
31732|         DevObj =
            | GetObjectFromDriveLetter(In->CacheFileName[12]);
31733|         if ( DevObj ) {
31734|             UNICODE_STRING LookingFor;
31735|
31736|             | RtlInitUnicodeString(&LookingFor,L"\\Driver\\Ftdisk");
31737|
31738|             // make sure ftdisk is NOT above us
31739|             // check to see if any one is attached to
            | this object
31740|             // going up the chain, as more than one
            | filter can be
31741|             // installed.
31742|
31743|             DevObj = DevObj->AttachedDevice;
31744|             while ( DevObj ) {
31745|                 if ( DevObj->DriverObject ) {
31746|
31747|                     // some one is layered on top of
                    | us.
31748|                     if (
                        | RtlCompareUnicodeString(&DevObj->DriverObject->DriverNam
                        | e,&LookingFor,TRUE)==0 ) {
31749|                         // hey its ftdisk
31750|                         Status = PSM_ERROR_DEADLOCK;
31751|                         break;
31752|                     }
31753|                     DevObj = DevObj->AttachedDevice;
31754|                 } else {
31755|                     // odd no driver associated with
                    | device.. oh well.
31756|                     break;
31757|                 }
31758|             }

```

```

31759|     } else {
31760|         // no device found for cache file..
31761|         // probally means sym link check failed
31762|         Status = STATUS_NO_SUCH_DEVICE;
31763|     }
31764| }
31765| #endif
31766| return Status;
31767| }
31768|
31769| NTSTATUS CloseCacheFilesThatAreNotBeingUsed()
31770| {
31771|     NTSTATUS Status = STATUS_SUCCESS;
31772|
31773|     __try {
31774|
31775|         | Debug(DEBUG_DCPSM,("CloseCacheFilesThatAreNotBeingUsed:
31776|         | Called\n"));
31777|         PDEVICE_OBJECT DevObj =
31778|         | PSMAN_DRIVER_OBJECT->DeviceObject;
31779|         while ( DevObj ) {
31780|             // if a filtered disk
31781|             if (
31782|                 | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
31783|                 PFILTERED_EXTENSION DevExt =
31784|                 | GetFilteredExtension(DevObj);
31785|                 if (
31786|                     | IsListEmpty(&DevExt->Cache.SnapShotHead) ) {
31787|                     | PersistentDictionary::CloseFilesForVolume(DevObj);
31788|                 }
31789|             } // if filtered disk
31790|             DevObj = DevObj->NextDevice;
31791|         } // while(DevObj)
31792|     } __except
31793|     | (ExceptionFilter(GetExceptionInformation())) {
31794|         | Debug(DEBUG_DCPSM,("CloseCacheFilesThreadAreNotBeingUsed
31795|         | : Exception %08x",GetExceptionCode()));
31796|     }
31797|
31798|     | Debug(DEBUG_DCPSM,("CloseCacheFilesThatAreNotBeingUsed:
31799|     | Done\n"));
31800|
31801|     return Status;
31802| }
31803|
31804| NTSTATUS OpenCacheFiles( pOpenTransactionInternal

```

```

    | In, PVOID *ObjectTable, ULONG NumObjects, PVOID
    | AbortEvent )
31797| {
31798|     NTSTATUS Status = STATUS_SUCCESS;
31799|
31800|     __try {
31801|         PDEVICE_OBJECT DevObj =
    | PSMAN_DRIVER_OBJECT->DeviceObject;
31802|         while ( DevObj ) {
31803|             // if a filtered disk
31804|             if (
    | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
31805|                 ULONG k;
31806|                 PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DevObj);
31807|
31808|                 for ( k=0;k<NumObjects;k++ ) {
31809|                     if ( &DevExt->WriteEvent ==
    | ObjectTable[k] ) {
31810|                         // we change the dev ext to be
    | true as we do have it acquired
31811|                         // otherwise if the volume
    | needs to be reloaded, we will get a deadlock
31812|                         ULONG Save =
    | DevExt->OpenCloseAcquired;
31813|                         DevExt->OpenCloseAcquired =
    | TRUE;
31814|                         // okay, found a volume that we
    | are snapping
31815|                         Status =
    | PersistentDictionary::RebuildSnapshotsForVolume(DevObj,F
    | ALSE,AbortEvent);
31816|                         DevExt->OpenCloseAcquired=Save;
31817|                         if ( !NT_SUCCESS(Status) ) {
31818|
31819|                             // cleanup objects we
    | already did
31820|                             PDEVICE_OBJECT D =
    | PSMAN_DRIVER_OBJECT->DeviceObject;
31821|                             while ( D ) {
31822|                                 // if a filtered disk
31823|                                 if (
    | PsmGetObjectTypes(D)==OBJECT_FILTEREDDISK ) {
31824|                                     ULONG o;
31825|                                     PFILTERED_EXTENSION
    | DE = GetFilteredExtension(D);
31826|                                     for(o=0;o<k;o++) {
31827|                                         if (
    | &DE->WriteEvent == ObjectTable[o] ) {
31828|                                             if (

```

```

    | DE->Cache.ReferenceCount > 0 ) {
31829|         if (
    | --DE->Cache.ReferenceCount==0 ) {
31830|
    | PersistentDictionary::TearDownCacheForVolume(D);
31831|         }
31832|     }
31833| }
31834| }
31835| }
31836|     D = D->NextDevice;
31837| }
31838|
31839|     return Status;
31840| }
31841| }
31842| }
31843| } // if filtered disk
31844|
31845|     DevObj = DevObj->NextDevice;
31846| } // while(DevObj)
31847| } __except
    | (ExceptionFilter(GetExceptionInformation())) {
31848|     Status = GetExceptionCode();
31849|     Debug(DEBUG_DCPSM,("OpenCacheFiles: Exception
    | %08x",Status));
31850| }
31851|
31852|     return Status;
31853| }
31854|
31855|
31856| void CleanupSignals( ULONG NumDrives, PVOID
    | *ObjectTable, BOOLEAN CleanupCache )
31857| {
31858|     PDEVICE_OBJECT DevObj =
    | PSMANDriverObject->DeviceObject;
31859|
31860|     while ( DevObj ) {
31861|         Debug(DEBUG_DCPSM,("CleanupSignals: Cleaning
    | up DevObj=%08x\n",DevObj));
31862|         // if a filtered disk
31863|         if (
    | PsmGetObjectype(DevObj)==OBJECT_FILTEREDDISK ) {
31864|             ULONG k;
31865|             PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DevObj);
31866|
31867|             for ( k=0;k<NumDrives;k++ ) {
31868|                 if ( &DevExt->WriteEvent ==

```

```

    | ObjectTable[k] ) {
31869|         InterlockedDecrement((PLONG)
    | &DevExt->SignalWrite);
31870|
31871|         if(CleanupCache) {
31872|
    | ASSERT(DevExt->Cache.ReferenceCount);
31873|         if (
    | --DevExt->Cache.ReferenceCount==0 ) {
31874|
    | PersistentDictionary::TearDownCacheForVolume(DevObj);
31875|         }
31876|     }
31877| }
31878| }
31879| } // if filtered disk
31880|
31881|     DevObj = DevObj->NextDevice;
31882| } // while(DevObj)
31883| }
31884|
31885|
31886| NTSTATUS CheckForAbortCreatingSnapShots ()
31887| {
31888|     NTSTATUS Status = STATUS_SUCCESS;
31889|     PDEVICE_OBJECT DevObj =
    | PSMAN_DRIVER_OBJECT->DeviceObject;
31890|
31891|     while ( DevObj && NT_SUCCESS(Status) ) {
31892|         // if a filtered disk
31893|         if (
    | PsmGetObjectType(DevObj)==OBJECT_FILTEREDDISK ) {
31894|             PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DevObj);
31895|             if ( DevExt->SignalWrite ) {
31896|                 GetSnapShotForRead();
31897|                 __try {
31898|                     pkSnapShotEntry p =
    | GetTopSnapShot(&DevExt->SnapShots);
31899|                     if ( p ) {
31900|                         __try {
31901|                             // There is at least one
    | snapshot on the volume already.
31902|                             // We only need to look at
    | one of the snapshots, because they will all say the
    | same thing
31903|                             // when asked about cache
    | usage.
31904|
31905|                             if (

```

```

    | ((pPersistentDictionary)(p->Dictionary))->IsCacheSnapSho
    | tCreationThresholdReached() ) {
31906|         Status =
    | PSM_INSUFFICIENT_CACHE; // will cause outer loop to
    | exit
31907|     }
31908|     } __finally {
31909|         DoneWithSnapShot(p);
31910|     }
31911| }
31912| } __finally {
31913|     ReleaseSnapShotForRead();
31914| }
31915| }
31916| } // if filtered disk
31917|
31918|     DevObj = DevObj->NextDevice;
31919| } // while(DevObj)
31920|
31921| return Status;
31922| }
31923|
31924| //-----
    | -----
31925| // GetSequenceForNewSnapShot - This function searches
    | the volumes currently
31926| // pending snapshot creation, finds the maximum
    | sequence number so far, and
31927| // adds one to get a value guaranteed to be larger
    | than any yet existing on
31928| // those volumes. This fixes the following problems:
31929| //
31930| // 1. Multi-volume revert at boot not working,
    | because sequence numbers don't match.
31931| //
31932| // 2. Multi-volume snapshots getting "de-linked"
    | after volume(s) remount.
31933| //
31934| NTSTATUS GetSequenceForNewSnapShot ( ULONG &NewSequence
    | )
31935| {
31936|     ULONG NumSequencesFound = 0;
31937|     NTSTATUS Status = STATUS_NOT_FOUND;
31938|
31939|     __try {
31940|         NewSequence = 0;
31941|         PDEVICE_OBJECT DevObj =
    | PSMAN_DRIVER_OBJECT->DeviceObject;
31942|         while ( DevObj ) {
31943|             if (

```

```

    | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
31944|         PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DevObj);
31945|         if ( DevExt->SignalWrite ) {
31946|             if ( DevExt->Cache.Header ) {
31947|                 ULONG ThisSequence =
    | DevExt->Cache.Header->HighestSnapNumber;
31948|                 ++NumSequencesFound;
31949|
    | Debug(DEBUG_DCPSM,("GetSequenceForNewSnapShot: Found
    | sequence %08x in DevExt %08x (found %08x so
    | far)\n",ThisSequence,DevExt,NumSequencesFound));
31950|             if ( ThisSequence >=
    | NewSequence ) {
31951|                 NewSequence = 1 +
    | ThisSequence;
31952|                 Status = STATUS_SUCCESS;
31953|             }
31954|         }
31955|     }
31956| }
31957|
31958|     DevObj = DevObj->NextDevice;
31959| }
31960| } __except
    | (ExceptionFilter(GetExceptionInformation())) {
31961|     Status = GetExceptionCode();
31962|     Debug(DEBUG_DCPSM,("!!!!
    | GetSequenceForNewSnapShot: Exception %08x\n",Status));
31963| }
31964|
31965|     Debug(DEBUG_PROCCALL,("GetSequenceForNewSnapShot
    | returning Status=%08x,
    | NewSequence=%08x\n",Status,NewSequence));
31966|     ASSERT (NewSequence > 0);
31967|     ASSERT (NumSequencesFound > 0);
31968|     ASSERT (Status == STATUS_SUCCESS);
31969|     return Status;
31970| }
31971|
31972| /*
31973| This routine will wait for the Quiescent window, and
    | if available will return with the devices being PSMed.
31974| */
31975| /*-----*/
    | -----*/
31976| NTSTATUS WaitForQuiescentPeriod( pOT_USER User,
    | tOpenTransactionInInternal *Buffer, tkSnapShotMaster
    | **MasterSnapShot, PKEVENT AbortEvent )
31977| {

```



```

31978|  PKWAIT_BLOCK WaitBlock=NULL;
31979|  PVOID      *ObjectTable=NULL;
31980|  ULONG NumDrives=0;
31981|  NTSTATUS   Status=STATUS_INSUFFICIENT_RESOURCES;
31982|  PFILTERED_EXTENSION DevExt=NULL;
31983|  PDEVICE_OBJECT DevObj=NULL;
31984|  pkSnapshotEntry p=NULL;
31985|
31986|  Debug(DEBUG_PROCCALL,("Entering
    | WaitForQPeriod\n"));
31987|  CurrentTime.QuadPart=0;
31988|
31989|  *MasterSnapShot = (tkSnapshotMaster *)
    | MemAllocatePoolWithTag( NonPagedPool,
    | sizeof(tkSnapshotMaster),PSM_MASTER_SNAPSHOT);
31990|  if ( *MasterSnapShot ) {
31991|      RtlZeroMemory( *MasterSnapShot,
    | sizeof(tkSnapshotMaster));
31992|
31993|      | InitializeListHead(&(*MasterSnapShot)->SnapShots);
31994|      (*MasterSnapShot)->ExclusiveProcess = 0;
31995|      (*MasterSnapShot)->DllPrivateUse =
    | Buffer->DllPrivateUse;
31996|      (*MasterSnapShot)->GroupNumber = (ULONG)
    | (Buffer->CallerPrivateUse);
31997|      (*MasterSnapShot)->NumToKeep =
    | Buffer->NumToKeep;
31998|      (*MasterSnapShot)->Priority = Buffer->Priority;
31999|      (*MasterSnapShot)->SnapShotFlags=
    | Buffer->SnapShotFlags;
32000|      // UserSnapShotName is set later in
    | SbSetUserName
32001|      if ( Buffer->InternalFlags &
    | PSM_IFLAG_PERSISTENT ) {
32002|          (*MasterSnapShot)->Persistent = TRUE;
32003|      } else {
32004|          (*MasterSnapShot)->Persistent = FALSE;
32005|      }
32006|
32007|      WaitBlock = (PKWAIT_BLOCK)
    | MemAllocatePoolWithTag( NonPagedPool,
    | MAXIMUM_WAIT_OBJECTS*sizeof(KWAIT_BLOCK),QTAG);
32008|      if ( WaitBlock ) {
32009|          RtlZeroMemory( WaitBlock,
    | MAXIMUM_WAIT_OBJECTS*sizeof(KWAIT_BLOCK));
32010|          ObjectTable = (PVOID *)
    | MemAllocatePoolWithTag( NonPagedPool,
    | MAXIMUM_WAIT_OBJECTS*sizeof(PVOID),QTAG);
32011|          if ( ObjectTable ) {

```

```

32012|         RtlZeroMemory( ObjectTable,
| MAXIMUM_WAIT_OBJECTS*sizeof(PVOID));
32013|
32014|         // make list of drives to psm
32015|         // it fills in ObjectTable for us.
32016|         Status = MakeVolumeListToPsm( Buffer,
| ObjectTable, &NumDrives);
32017|
32018|         if ( (!Status) && (NumDrives>0) ) {
32019|
| UpdateGlobalStatus(PSM_WAITING_FOR QUIESCENT_PERIOD);
32020|         Status =
| OpenCacheFiles(Buffer,ObjectTable,NumDrives,AbortEvent);
32021|         if ( NT_SUCCESS(Status) ) {
32022|             LARGE_INTEGER
| Timeout,TimeToQuit;
32023|
32024|
| UpdateGlobalStatus(PSM_WAITING_FOR QUIESCENT_PERIOD);
32025|
32026|             ObjectTable[NumDrives] =
| AbortEvent;
32027|
32028|             Status=STATUS_SUCCESS;
32029|
32030|             // do a flush before we start
| looking for our window.
32031|             FlushVolumeList(Buffer);
32032|
32033|             // store for GetProgress
32034|             SecondsToWait =
| Buffer->QuiescentWait;
32035|
32036|             Debug(DEBUG_DCPSM,("WaitForQ:
| Quiescent Wait=%d,
| timeout=%d\n",Buffer->QuiescentWait,Buffer->QuiescentTim
| eout));
32037|
32038|             KeQuerySystemTime(&TimeToQuit);
32039|
32040|             // add how many seconds to wait
| for
32041|             TimeToQuit.QuadPart +=
| SECONDS(((signed long)Buffer->QuiescentTimeout));
32042|             (*MasterSnapShot)->OutOfSeconds
| = Buffer->QuiescentWait;
32043|
32044|             WaitLoop:
32045|             // do a flush before we start
| looking for our window.

```

```

32046|          FlushVolumeList(Buffer);
32047|
32048|          // if not waiting for any q
          | time, set status to timeout so it follows
32049|          // the right logic
32050|          // we are not doing a goto
          | because of the __try stack frame and i do not know
32051|          // if its set up correctly if
          | called into it.
32052|          if ( Buffer->QuiescentWait!=0 )
          | {
32053|          // get current time
32054|
          | KeQuerySystemTime(&CurrentTime);
32055|          Timeout.QuadPart =
          | RELATIVE(SECONDS(((signed
          | long)Buffer->QuiescentWait)));
32056|          Status = STATUS_SUCCESS;
32057|
32058|          // wait while not a
          | timeout, not time to quit yet, and not exit event.
32059|          while (
          | (Status!=STATUS_TIMEOUT) &&
32060|          | (CurrentTime.QuadPart<TimeToQuit.QuadPart) &&
32061|          | ((ULONG)Status!=NumDrives) ) {
32062|
32063|          | //Debug(DEBUG_DCPSPM,("WaitForQ: (1) before
          | KeWaitForMultipleObjects\n"));
32064|          Status =
          | KeWaitForMultipleObjects(
32065|          | AbortEvent ? NumDrives+1 : NumDrives, //IN ULONG Count,
32066|          | ObjectTable, //IN PVOID Object[],
32067|          | WaitAny, //IN WAIT_TYPE WaitType,
32068|          | Executive, //IN KWAIT_REASON WaitReason,
32069|          | (KPROCESSOR_MODE)KernelMode, //IN KPROCESSOR_MODE
          | WaitMode,
32070|          | FALSE, //IN BOOLEAN Alertable,
32071|          | &Timeout, //IN PLARGE_INTEGER Timeout OPTIONAL,
32072|          | WaitBlock //IN PKWAIT_BLOCK WaitBlockArray OPTIONAL

```

```

32073|
| );
32074|
| //Debug(DEBUG_DCPSM,("WaitForQ: (1) after
| KeWaitForMultipleObjects\n"));
32075|
| KeQuerySystemTime(&CurrentTime);
32076|         if (
| (Status!=STATUS_TIMEOUT) &&
| ((ULONG)Status!=NumDrives+1) ) {
32077|             NTSTATUS AStatus =
| CheckForAbortCreatingSnapShots();
32078|             if ( AStatus!=0 ) {
32079|                 Status =
| AStatus;
32080|                 break;
32081|             }
32082|         }
32083|     }
32084| } else {
32085|     Status = STATUS_TIMEOUT;
32086| }
32087|
32088| if ( Status==STATUS_TIMEOUT ) {
32089|
| //Debug(DEBUG_DCPSM,("WaitForQ: (1) Got the wait
| time!\n"));
32090|         // we got the wait time!
32091|         // now try to turn on psm
| before any more ios happen.
32092|
32093|         // make sure we flush since
| writes could still be in memory,
32094|         // but hasnt reached us yet
| due to NTs lazy writer.
32095|         // no need to do this as
| the above flush forced the disk to be in sync,
32096|         // and we do not care about
| writes that havent affect the disk yet.
32097|         // 8-5-99
32098|         //
| FlushVolumeList(Buffer);
32099|
32100|         // get the global device to
| stop read and write requests
32101|         // this is done because a
| write could have occurred by the
32102|         // time we got to this
| point, and before we can traverse
32103|         // the list of devices to

```

```

    | PSM to turn them on.
32104|                // we do not want 1 volume
    | PSMed at a different time than
32105|                // another, or have a
    | couple extra ios sent to it.
32106|
32107|                // get snapshot before
    | global or it causes a deadlock
32108|
    | //Debug(DEBUG_DCPSM,("WaitForQ: Before
    | GetSnapShotForWrite\n",Status));
32109|                GetSnapShotForWrite();
32110|
    | //Debug(DEBUG_DCPSM,("WaitForQ: Before
    | GetGlobalDeviceForWrite\n",Status));
32111|                GetGlobalDeviceForWrite ();
32112|
    | //Debug(DEBUG_DCPSM,("WaitForQ: After
    | GetGlobalDeviceForWrite\n",Status));
32113|
32114|                __try {
32115|                // a zero means just
    | check
32116|                Timeout.QuadPart = 0;
32117|
    | if (
    | Buffer->QuiescentWait!=0 ) {
32119|                // Okay check to
    | see if any writes occurred AFTER we got the window
32120|
    | //Debug(DEBUG_DCPSM,("WaitForQ: (2) before
    | KeWaitForMultipleObjects\n"));
32121|                Status =
    | KeWaitForMultipleObjects(
32122|
    | NumDrives, //IN ULONG Count,
32123|
    | ObjectTable, //IN PVOID Object[],
32124|
    | WaitAny, //IN WAIT_TYPE WaitType,
32125|
    | Executive, //IN KWAIT_REASON WaitReason,
32126|
    | (KPROCESSOR_MODE)KernelMode, //IN KPROCESSOR_MODE
    | WaitMode,
32127|
    | FALSE, //IN BOOLEAN Alertable,
32128|
    | &Timeout, //IN PLARGE_INTEGER Timeout OPTIONAL,
32129|

```

```

    | WaitBlock //IN PKWAIT_BLOCK WaitBlockArray OPTIONAL
32130|
    | );
32131|
    | //Debug(DEBUG_DCPSM,("WaitForQ: (2) after
    | KeWaitForMultipleObjects\n"));
32132|                }
32133|
32134|                if (
    | Status==STATUS_TIMEOUT ) {
32135|
    | //Debug(DEBUG_DCPSM,("WaitForQ: (2) Got the wait
    | time!\n"));
32136|                PDEVICE_OBJECT
    | CacheFileVolume;
32137|                LARGE_INTEGER
    | PSMTime;
32138|
32139|                // YES! we got the
    | window! we still have io locked out!
32140|                // go ahead and psm
    | the devices.
32141|
32142|                // store the time
    | so we can mark the volumes with the same time
32143|
    | KeQuerySystemTime(&PSMTime);
32144|
32145| #if 0 // This code is a relic from single snapshot
    | days. (eg.1.02?). It's purpose then was to test
    | cachefile growability.
32146|                // It's maybe had a
    | bug since pre-persistence multi-snapshot days (eg
    | 1.10?). Since input param was just ignored after
    | first snapshot had assigned cachefile.
32147|                // Now failing
    | continually (2.00). Because the parameter space is
    | used for something else (Group name) !!!
32148|
32149|                // BUT.... before
    | removing it, consider if it needs fixing to make
    | temporary multi-snapshot work!!!!!!!
32150|
32151|                // get device
    | object cache file resides on
32152|                CacheFileVolume =
    | GetObjectFromWin32Name(Buffer->CacheFileName);
32153|
    | ASSERT(CacheFileVolume);
32154| #endif

```

```

32155|
32156|             ULONG
    | NextSnapShotSequence = 0;
32157|
    | GetSequenceForNewSnapShot (NextSnapShotSequence);
    | //FIXFIXFIX: check for returned error
32158|
32159|             DevObj =
    | PSMANDriverObject->DeviceObject;
32160|             while ( DevObj ) {
32161|
    | //Debug(DEBUG_DCPSM,("WaitForQ:
    | DevObj=%08x\n",DevObj));
32162|             // if a
    | filtered disk
32163|             if (
    | PsmGetObjectype(DevObj)==OBJECT_FILTEREDDISK ) {
32164|             DevExt =
    | GetFilteredExtension(DevObj);
32165|
    | if (
    | DevExt->SignalWrite ) {
32167|             //
    | these should be open if we are creating a snapshot
32168|
    | ASSERT(IsValidHandle(DevExt->Cache.HeaderFile.FileHandle
    | ));
32169|
    | ASSERT(IsValidHandle(DevExt->Cache.IndexFile.FileHandle)
    | );
32170|
    | ASSERT(IsValidHandle(DevExt->Cache.CacheFile.FileHandle)
    | );
32171|
32172|             p =
    | (tkSnapShotEntry *) MemAllocatePoolWithTag(
    | NonPagedPool,
    | sizeof(tkSnapShotEntry),PSM_SNAPSHOT_ENTRY);
32173|
    | //FIXFIXFIX check and back out if p==null
32174|
    | ASSERT(p);
32175|
32176|
    | RtlZeroMemory (p, sizeof(tkSnapShotEntry));
32177|
    | p->DeviceObject = DevObj;
32178|
    | p->MasterSnapShot = (*MasterSnapShot);
32179|

```

```

    | p->Deleted = FALSE;
32180|
    | p->Count=0;
32181|
    | p->PSMSectors = NULL;
32182|
    | (*MasterSnapShot)->SnapShotTime=PSMTime;
32183|
32184|                                     if (
    | Buffer->InternalFlags & PSM_IFLAG_PERSISTENT ) {
32185|
    | Debug(DEBUG_DCPSM,("WaitForQ: Before Dictionary::Open
    | (persistent)\n"));
32186|
32187|
    | Status = Dictionary::Open(
32188|
    | DICT_FLAG_PERSISTENT,
32189|
    | p->Dictionary );
32190|
32191|                                     if
    | ( Status == STATUS_SUCCESS ) {
32192|
    | ASSERT(p->Dictionary);
32193|
    | //Debug(DEBUG_DCPSM,("WaitForQ: Before pd::SetVolume
    | (persistent)\n"));
32194|
    | ((pPersistentDictionary)p->Dictionary)->SetVolume(
32195|
    | DevObj,
32196|
    | *MasterSnapShot,
32197|
    | NextSnapShotSequence );
32198|                                     }
32199|                                     } else
    | {
32200|
    | Debug(DEBUG_DCPSM,("WaitForQ: Before Dictionary::Open
    | (temporary)\n"));
32201|
    | Status = Dictionary::Open(
32202|
    | DICT_FLAG_NONPERSISTENT,
32203|
    | p->Dictionary );
32204|
32205|                                     if

```



```

    | ( Status == STATUS_SUCCESS ) {
32206|
    | ASSERT(p->Dictionary);
32207|
    | //Debug(DEBUG_DCPSM,("WaitForQ: Before pd::SetVolume
    | (temporary)\n"));
32208|
    | ((pTemporaryDictionary)p->Dictionary)->SetVolume (
32209|
    | DevObj,
32210|
    | *MasterSnapShot,
32211|
    | NextSnapShotSequence );
32212|         }
32213|     }
32214|
32215|         if (
    | Status == STATUS_SUCCESS ) {
32216|
    | InterlockedIncrement((PLONG)&(*MasterSnapShot)->Count);
32217|
32218|
    | InterlockedDecrement((PLONG) &DevExt->SignalWrite);
32219|
    | InterlockedIncrement((PLONG) &DevExt->PSMed);
32220|
    | InterlockedIncrement((PLONG) &DevExt->OpenCount);
32221|
32222|
    | InsertHeadList(&DevExt->SnapShots,&p->DevExt);
32223|
    | InsertHeadList(&(*MasterSnapShot)->SnapShots,&p->Master)
    | ;
32224|
    | InsertHeadList(&User->SnapShots,&p->User);
32225|
    | //AddDeviceToList(User->SnapShots,p);
32226|
    | InterlockedIncrement((PLONG) &User->NumOpenSnapShots);
32227|
    | Debug(DEBUG_DICT,("Created dictionary %08x for object
    | %08x\n",p->Dictionary,DevExt->DeviceObject));
32228|         } else
    | {
32229|
    | Debug(DEBUG_DCPSM,("WaitForQ: !!! Dictionary creation
    | error; Status=%08x\n",Status));
32230|         }
32231|     }

```

```

32232|                } // if
    | filtered disk
32233|
32234|                DevObj =
    | DevObj->NextDevice;
32235|                } // while(DevObj)
32236|
32237|                Status =
    | STATUS_SUCCESS;
32238|                } else {
32239|
    | Debug(DEBUG_DCPSM,("WaitForQ: Write occurred before we
    | could lock out I/O\n"));
32240|                // damn, a write
    | occurred before we could lock out io.
32241|                // try again.
32242|                // STATUS_WAIT_0 ==
    | STATUS_SUCCESS, so change to something else
32243|                Status =
    | STATUS_TIMEOUT;
32244|                }
32245|                } __finally {
32246|                // make sure this
    | always get released as all IO is hung up at this point
32247|
    | ReleaseGlobalDeviceForWrite();
32248|
    | ReleaseSnapShotForWrite();
32249|
    | //Debug(DEBUG_DCPSM,("WaitForQ: After
    | ReleaseGlobalDeviceForWrite and
    | ReleaseSnapShotForWrite\n"));
32250|                }
32251|
32252|                // can not do goto in a
    | try..finally block.
32253|                if ( Status!=STATUS_SUCCESS
    | ) {
32254|
    | //Debug(DEBUG_DCPSM,("WaitForQ: (1) goto
    | WaitLoop\n"));
32255|                goto WaitLoop;
32256|                }
32257|
    | } else {
32258|                } else {
32259|                if (
    | (int)Status==(int)(STATUS_WAIT_0+NumDrives) ) {
32260|                // our exit event was
    | signaled! this gets convert to ERROR_ACCESS_DENIED
32261|                // who did this

```

```

    | conversion chart anyways , sheesh.
32262|             Status =
    | PSM_CANCELED_BY_USER;
32263|             } else {
32264|             // hmm we have been
    | waited the timeout period, return failure
32265|             // STATUS_TIMEOUT is
    | not an error condition.
32266|
    | Debug(DEBUG_DCPSM,("Error %08x on wait!!!\n",Status));
32267|
32268|             if ( Status !=
    | PSM_INSUFFICIENT_CACHE ) {
32269|                 if ( Buffer->Flags
    | & PSM_FLAG_FORCE_SNAPSHOT ) {
32270|                     Status = 0;
32271|
    | (*MasterSnapShot)->Status = PSM_SNAPSHOT_FORCED;
32272|
    | Buffer->QuiescentWait = 0;
32273|
    | //Debug(DEBUG_DCPSM,("WaitForQ: (2) goto
    | WaitLoop\n"));
32274|                 goto WaitLoop;
32275|             }
32276|
32277|             Status =
    | PSM_ERROR_TIMEOUT;
32278|         }
32279|     }
32280| }
32281|
32282|     if ( !INT_SUCCESS(Status) ) {
32283|         // free signals so we dont
    | accidentally use them when called again
32284|
32285|         // FIXFIXFIX free
    | snapshots!
32286|         // FIXFIXFIX This !assumes!
    | that NO devices
32287|         // were successful. (ie
    | timeout only, not out of memory)
32288|
    | CleanupSignals(NumDrives,ObjectTable,TRUE);
32289|     }
32290| } else {
32291|     Debug(DEBUG_DCPSM,("Error %08x
    | opening files for volumes!!!\n",Status));
32292|     //ASSERT(FALSE);
32293|

```

```

    | CleanupSignals(NumDrives,ObjectTable,FALSE);
32294|     }
32295|     } else {
32296|         if ( NumDrives==0 ) {
32297|             Status = STATUS_NO_SUCH_DEVICE;
32298|         }
32299|     }
32300|     FREE_POINTER(ObjectTable);
32301| }
32302|
32303|     FREE_POINTER(WaitBlock);
32304| }
32305|
32306|     if ( !NT_SUCCESS(Status) ) {
32307|         FREE_POINTER(*MasterSnapShot);
32308|     }
32309| }
32310|     CurrentTime.QuadPart = 0;
32311|     //Debug(DEBUG_DCPSM,("WaitForQ: Before
    | CloseCacheFilesThatAreNotBeingUsed\n"));
32312|     CloseCacheFilesThatAreNotBeingUsed();
32313|     UpdateGlobalStatus(PSM_IDLE);
32314|     Debug(DEBUG_DCPSM,("WaitForQ: returning
    | %08x\n",Status));
32315|     return Status;
32316| }
32317|
32318| /*-----
    | -----*/
32319| NTSTATUS DelDeviceFromList( PLIST_ENTRY List,
    | pkSnapShotEntry SnapShot)
32320| {
32321|     pkSnapShotEntry p;
32322|     PLIST_ENTRY ListEntry;
32323|
32324| #ifdef DEBUG
32325|     if ( !IsSnapShotAcquiredForWrite() ) {
32326|         Debug(DEBUG_DCPSM,("DelDeviceToList: Snapshot
    | resource not acquired!\n"));
32327|         DbgBreakPoint();
32328|     }
32329| #endif
32330|
32331|     if ( IsListEmpty(List) )
32332|         return STATUS_UNSUCCESSFUL;
32333|
32334|     ListEntry = List->Flink;
32335|
32336|     while ( ListEntry!=List ) {
32337|         /*lint -save -e413 */

```

```

32338|     p = CONTAINING_RECORD( ListEntry,
    | tkSnapShotEntry, User);
32339|     /*lint -restore */
32340|
32341|     if ( p==SnapShot ) {
32342|         RemoveEntryList(&SnapShot->User)
32343|         return STATUS_SUCCESS;
32344|     }
32345|     ListEntry = ListEntry->Flink;
32346| }
32347| return STATUS_UNSUCCESSFUL;
32348| }
32349|
32350|
32351| /*-----
    | -----*/
32352| NTSTATUS FreePSMVolume( pOT_USER User, PDEVICE_OBJECT
    | DeviceObject, tkSnapShotEntry *SnapShot )
32353| {
32354|     PFILTERED_EXTENSION DevExt=NULL;
32355|     NTSTATUS Status=STATUS_SUCCESS;
32356|
32357|     Debug(DEBUG_PROCCALL,("FreePSMVolume called
    | user=%08x do=%08x
    | ss=%08x\n",User,DeviceObject,SnapShot));
32358| #ifdef DEBUG
32359|     if ( !IsSnapShotAcquiredForWrite() ) {
32360|         Debug(DEBUG_DCPSM,("FreePSMVolume: Snapshot
    | resource not acquired!\n"));
32361|         DbgBreakPoint();
32362|     }
32363|     if ( !DeviceObject ) {
32364|         Debug(DEBUG_DCPSM,("FreePSMVolume:
    | DeviceObject==NULL!\n"));
32365|         DbgBreakPoint();
32366|     }
32367| #endif
32368|
32369|
32370|     __try {
32371| #if 1
32372|         // TESTTEST
32373|         if ( DeviceObject ) {
32374|             DevExt =
    | GetFilteredExtension(DeviceObject);
32375|             Debug(DEBUG_DCPSM,("FreePSMVolume: Freeing
    | %S!\n",DevExt->Name));
32376|             if ( SnapShot ) {
32377|                 if ( User ) {
32378|

```

```

    | ASSERT(&SnapShot->User!&User->SnapShots);
32379|         if (
    | DelDeviceFromList(&User->SnapShots,SnapShot)==STATUS_SUC
    | CESS ) {
32380|             if ( User->NumOpenSnapShots ) {
32381|
    | InterlockedDecrement((PLONG) &User->NumOpenSnapShots);
32382|             } else {
32383|                 // FIXFIXFIX can this
    | happen?
32384|
    | Debug(DEBUG_DCPSM,("FreePSMVolume: Doesnt have psm
    | active for that volume!\n"));
32385| #ifdef DEBUG
32386|                 DbgBreakPoint();
32387| #endif
32388|             }
32389|         } else {
32390|             // This happens when virtual
    | volumes havent been mapped in yet
32391|
    | Debug(DEBUG_DCPSM,("FreePSMVolume: Unable to delete
    | from list!\n"));
32392|         }
32393|     }
32394|     tkSnapShotMaster
    | *Master=SnapShot->MasterSnapShot;
32395|
32396|
    | FreeResourcesForVolume(DeviceObject,SnapShot);
32397|
32398|         // SnapShot "could" be free by the time
    | FreeResourcesForVolume returns.
32399|         Debug(DEBUG_DCPSM,("FreePSMVolume:
    | Freeing resources for volume!\n"));
32400|         // volume not in use.. free
32401|         TdDelDrive(DeviceObject,Master);
32402|     }
32403| }
32404| } __except
    | (ExceptionFilter(GetExceptionInformation())) {
32405|     Status = GetExceptionCode();
32406|     Debug(DEBUG_DCPSM,("FreePsmVolume: Exception
    | %08x\n",Status));
32407| }
32408|
32409| #endif
32410| return Status;
32411| }
32412|

```

```

32413| /*-----
| -----*/
32414| NTSTATUS IsInUserList( pOT_USER User, pkSnapshotEntry
| Snapshot)
32415| {
32416|     pkSnapshotEntry p;
32417|     PLIST_ENTRY ListEntry;
32418| #ifdef DEBUG
32419|     if ( (!IsSnapshotAcquiredForRead()) &&
| (!IsSnapshotAcquiredForWrite()) ) {
32420|         Debug(DEBUG_DCPSM,("IsInUserList: Snapshot
| resource not acquired!\n"));
32421|         DbgBreakPoint();
32422|     }
32423| #endif
32424|
32425|     if ( IsListEmpty(&User->SnapShots) )
32426|         return STATUS_UNSUCCESSFUL;
32427|
32428|     ListEntry = User->SnapShots.Flink;
32429|
32430|     while ( ListEntry!=&User->SnapShots ) {
32431|         /*lint -save -e413 */
32432|         p = CONTAINING_RECORD( ListEntry,
| tkSnapshotEntry, User);
32433|         /*lint -restore */
32434|
32435|         if ( p==Snapshot ) {
32436|             return STATUS_SUCCESS;
32437|         }
32438|         ListEntry = ListEntry->Flink;
32439|     }
32440|     return STATUS_UNSUCCESSFUL;
32441| }
32442| /*-----
| -----*/
32443| /*
32444|     Frees all volumes for a specific snapshot
32445| */
32446| STATIC NTSTATUS
| GetRidOfSpecificSnapshotForUser(pOT_USER User,
| pkSnapshotMaster MasterSnapshot)
32447| {
32448|     NTSTATUS Status=STATUS_SUCCESS;
32449|     pkSnapshotEntry p;
32450|
32451| #ifdef DEBUG
32452|     if ( !IsSnapshotAcquiredForWrite() ) {
32453|         Debug(DEBUG_DCPSM,("GetRidOfSpecificSnapshotForUser:

```

```

    | Snapshot resource not acquired!\n"));
32454|     DbgBreakPoint();
32455| }
32456| #endif
32457|
32458|
    | p=GetTopSnapShotForMaster(&MasterSnapShot->SnapShots);
32459|     while ( p ) {
32460|         if ( IsInUserList(User,p)==STATUS_SUCCESS ) {
32461|             Status =
    | FreePSMVolume(User,p->DeviceObject,p);
32462|             // Start bak at top since a snapshot has
    | been deleted from this master
32463|             DoneWithSnapShot(p);
32464|
    | p=GetTopSnapShotForMaster(&MasterSnapShot->SnapShots);
32465|         } else {
32466|
    | Debug(DEBUG_DCPSM,("GetRifOfSpecificSnapShotForUser:
    | Snapshot %08x not in user list for master
    | %08x\n",p,MasterSnapShot));
32467| #ifdef DEBUG
32468|         DbgBreakPoint();
32469| #endif
32470|
    | p=GetNextSnapShotForMaster(&MasterSnapShot->SnapShots,p)
    | ;
32471|     }
32472| }
32473|
32474| return Status;
32475| }
32476|
32477| /*-----
    | -----*/
32478| /*
32479|     Frees ALL snapshots the user has open
32480| */
32481| STATIC NTSTATUS FreeVolumesForUser( pOT_USER User,
    | pkSnapShotMaster MasterSnapShot )
32482| {
32483|     NTSTATUS Status=STATUS_SUCCESS;
32484|     pkSnapShotEntry p;
32485|     PLIST_ENTRY ListEntry;
32486|
32487|     __try {
32488|         Debug(DEBUG_PROCCALL,("FreeVolumesForUser
    | called\n"));
32489|         GetSnapShotForWrite();
32490|         __try {

```



```

32491|         ListEntry = User->SnapShots.Flink;
32492|         while ( ListEntry!=&User->SnapShots ) {
32493|             /*lint -save -e413 */
32494|             p = CONTAINING_RECORD( ListEntry,
| tkSnapShotEntry, User);
32495|             /*lint -restore */
32496|
32497|             if ( p ) {
32498|                 // if a filtered disk
32499|                 if (
| PsmGetObject(p->DeviceObject)==OBJECT_FILTEREDDISK
| ) {
32500|                     PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(p->DeviceObject);
32501|                     if (
| InList(&DevExt->SnapShots,p) ) {
32502|
| Debug(DEBUG_DCPSM,("FreeVolumesForUser: Freeing device
| %08x!\n",p->DeviceObject));
32503|                     Status =
| FreePSMVolume(User,p->DeviceObject,p);
32504|                     // start back at top
32505|                     ListEntry =
| User->SnapShots.Flink;
32506|                     } else {
32507|
| Debug(DEBUG_DCPSM,("FreeVolumesForUser: device %08x
| belongs to snapshot %08x but it doesnt think
| so!\n",p->DeviceObject,p));
32508| #ifdef DEBUG
32509|                     DbgBreakPoint();
32510| #endif
32511|                     ListEntry =
| ListEntry->Flink;
32512|                     }
32513|                     } // if filtered disk
32514|                     } else {
32515|                         // this shouldnt happen!
32516|                         Debug(DEBUG_DCPSM,("User has null
| snapshot!!!!\n"));
32517| #ifdef DEBUG
32518|                         DbgBreakPoint();
32519| #endif
32520|                     }
32521|                     }
32522|                     } __finally {
32523|                         ReleaseSnapShotForWrite();
32524|                     }
32525|                 } __except
| (ExceptionFilter(GetExceptionInformation())) {

```

```

32526|     Status = GetExceptionCode();
32527|     Debug(DEBUG_DCPDM,("FreeVolumesForUser:
| Exception %08x, User %08x,
| Master=%08x\n",Status,User,MasterSnapShot));
32528| }
32529| return Status;
32530| }
32531|
32532| LIST_ENTRY
| CreationListHead={&CreationListHead,&CreationListHead};
32533|
32534| NTSTATUS CancelCreationOfSnapShots( )
32535| {
32536|     PLIST_ENTRY ListEntry;
32537|     ULONG Found=0;
32538|
32539|     pmAcquireMutex ( &PSMUserMutex, NULL );
32540|     __try {
32541|         ListEntry = CreationListHead.Flink;
32542|         while ( ListEntry!=&CreationListHead ) {
32543|             pAbortEventList
| Abort=CONTAINING_RECORD(ListEntry,tAbortEventList,ListEn
| try);
32544|             if ( Abort->AbortEvent ) {
32545|                 pmSetEvent(Abort->AbortEvent);
32546|                 Found++;
32547|             }
32548|             ListEntry = ListEntry->Flink;
32549|         }
32550|     } __finally {
32551|         pmReleaseMutex ( &PSMUserMutex);
32552|     }
32553|     return Found ? STATUS_SUCCESS : STATUS_NOT_FOUND;
32554| }
32555|
32556| /*-----
| -----*/
32557| void SbOpenPsmThread( pOpenPsmThread Thread )
32558| {
32559|     NTSTATUS Status = STATUS_UNSUCCESSFUL;
32560|     PKEVENT AbortEvent=NULL;
32561|     tAbortEventList Abort;
32562|
32563|     Debug(DEBUG_PROCCALL,("PSM Open Thread called %08x
| %08x
| %08x\n",Thread->User->ProcessID,Thread->User->ThreadID,T
| hread->User->FileObject));
32564|
32565|     __try {
32566|         ULONG InternalFlags =

```

```

| ((tOpenTransactionInternal
| *)Thread->OTI)->InternalFlags;
32567|     ULONG PersistentFlag = InternalFlags &
| PSM_IFLAG_PERSISTENT;
32568|     ULONG SaveTempFlag = InternalFlags &
| PSM_IFLAG_SAVE_TEMP_ON_EXIT;
32569|
32570|     if ( PersistentFlag || SaveTempFlag ) {
32571|         // Getting here means that we want to keep
| the snapshot after the invoking thread exits.
32572|         pOT_USER U =
| FindPSMUser(PsGetCurrentProcess(),(_ETHREAD*)-2);
32573|         if ( !U ) {
32574|             // internal system user hasnt been
| created, lets create it
32575|             U =
| InternalCreateUser(PsGetCurrentProcess(),(_ETHREAD*)-2,N
| ULL);
32576|         }
32577|
32578|         AbortEvent = Thread->User->AbortEvent;
32579|         Thread->User->AbortEvent = NULL;
32580|
32581|         if ( Thread->User->ErrorEvent ) {
32582|             // we dont use this, so get rid of
| reference
32583|             ObDereferenceObject(Thread->User->ErrorEvent);
32584|             Thread->User->ErrorEvent = NULL;
32585|         }
32586|         Thread->User = U;
32587|         Thread->User->Persistent = PersistentFlag ?
| TRUE : FALSE;
32588|         Thread->User->SaveTempOnExit = SaveTempFlag
| ? TRUE : FALSE;
32589|
32590|         if ( AbortEvent ) {
32591|             Abort.AbortEvent = AbortEvent;
32592|             pmAcquireMutex ( &PSMUserMutex, NULL
| );
32593|
| InsertHeadList(&CreationListHead,&Abort.ListEntry);
32594|             pmReleaseMutex ( &PSMUserMutex);
32595|         }
32596|     }
32597|
32598|     // copy In structure as it is the same as the
| out
32599|     pOpenTransactionOutInternal Out =
| (pOpenTransactionOutInternal)

```

```

    | MemAllocatePoolWithTag(PagedPool,Thread->OTOSize,TEMPTAG
    | );
32600|     if ( !Out ) {
32601|         Status = STATUS_INSUFFICIENT_RESOURCES;
32602|         goto Cleanup;
32603|     }
32604|
32605|     memset(Out,0,Thread->OTOSize);
32606|
32607|     __try {
32608|         Status =
            | SbOpenPSM(Thread->User,(tOpenTransactionInInternal
            | *)Thread->OTI,Thread->OTOSize,Out,AbortEvent);
32609|     }
            | __except(ExceptionFilter(GetExceptionInformation())) {
32610|         Status = GetExceptionCode();
32611|         Debug(DEBUG_DCPSM,("SbOpenPsm generated
            | exception %08x\n",Status));
32612|     }
32613|     memmove(Thread->OTI,Out,Thread->OTOSize);
32614|     FREE_POINTER(Out);
32615|     Cleanup:
32616|     __try {
32617|         if ( AbortEvent ) {
32618|             pmAcquireMutex ( &PSMUserMutex, NULL
            | );
32619|             // dont need this any more as its only
            | valid during open
32620|             RemoveEntryList(&Abort.ListEntry);
32621|             ObDereferenceObject(AbortEvent);
32622|             AbortEvent = NULL;
32623|             pmReleaseMutex ( &PSMUserMutex);
32624|         }
32625|     } __finally {
32626|         // make sure Irp is completed
32627|         Thread->Irp->IoStatus.Status = Status;
32628|
32629|         if ( NT_SUCCESS(Status) ) {
32630|             Thread->Irp->IoStatus.Information =
            | Thread->OTOSize;
32631|             IoCompleteRequest (Thread->Irp,
            | IO_DISK_INCREMENT) ;
32632|         } else {
32633|             Thread->Irp->IoStatus.Information = 0;
32634|             IoCompleteRequest (Thread->Irp,
            | IO_NO_INCREMENT) ;
32635|         }
32636|
32637|         FREE_POINTER(Thread);
32638|     }

```

```

32639|    }
    | __except(ExceptionFilter(GetExceptionInformation())) {
32640|        Status = GetExceptionCode();
32641|        Debug(DEBUG_DCPSM,("SbOpenPsmThread Exception
    | %08x\n",Status));
32642|    }
32643|
32644|    Debug(DEBUG_DCPSM,("SbOpenPsmThread done
    | %08x\n",Status));
32645|    PsTerminateSystemThread( 0 );
32646| }
32647|
32648| /*-----
    | -----*/
32649| NTSTATUS DeInitForExclusive ( pOT_USER User,
    | pkSnapShotMaster Master )
32650| {
32651|     pkSnapShotEntry p;
32652|
32653|     NOT_REFERENCED(User);
32654| #if 1
32655| // TESTTEST
32656|     NTSTATUS MasterValid =
    | ValidateKernelSnapShotPointer(Master);
32657|
32658|     if ( MasterValid == STATUS_SUCCESS ) {
32659|         GetSnapShotForWrite();
32660|         __try {
32661|
    | p=GetTopSnapShotForMaster(&Master->SnapShots);
32662|             while ( p ) {
32663|                 if ( p->PSMSectors ) {
32664|                     FREE_POINTER(p->PSMSectors);
32665|                 }
32666|
    | p=GetNextSnapShotForMaster(&Master->SnapShots,p);
32667|             } // while(p)
32668|         } __finally {
32669|             ReleaseSnapShotForWrite();
32670|         }
32671|     }
32672| #endif
32673|     return STATUS_SUCCESS;
32674| }
32675|
32676| /*-----
    | -----*/
32677| NTSTATUS InitForExclusive( pOT_USER User,
    | pkSnapShotMaster Master )
32678| {

```

```

32679|    ULARGE_INTEGER PL;
32680|    ULONG Remainder;
32681|    ULONG Count;
32682|    PFILTERED_EXTENSION DevExt=NULL;
32683|    NTSTATUS Status=STATUS_SUCCESS;
32684|    pkSnapshotEntry p;
32685|    ULONG NumSectors;
32686|
32687| #if 1
32688| // TESTTEST
32689|    GetSnapshotForWrite();
32690|    __try {
32691|        p=GetTopSnapshotForMaster(&Master->SnapShots);
32692|        while ( p ) {
32693|            if ( !IsInUserList(User,p)==STATUS_SUCCESS )
32694|                | {
32695|                    DevExt =
32696|                    | GetFilteredExtension(p->DeviceObject);
32697|                    // if psmmed, alloc memory for each
32698|                    | sector on the disk.
32699|                    if ( DevExt->PSMed ) {
32700|                        PL.QuadPart =
32701|                        | DevExt->Pi.PartitionLength.QuadPart;
32702|                        Debug(DEBUG_DCPSM,("Dcpsm:
32703|                        | InitForExclusive: Size of disk in bytes is
32704|                        | %!64d\n",PL.QuadPart));
32705|                        NumSectors =
32706|                        | RtlEnlargedUnsignedDivide ( PL, DevExt->BytesPerSector,
32707|                        | &Remainder);
32708|                        // dword align buffer or
32709|                        | RtlSetAllBits will corrupt past the end of the buffer
32710|                        Count = ((NumSectors+31) / 32) * 4;
32711|                        // allocate space for sector list
32712|                        | now that we know we have a device to psm.
32713|                        p->PSMSectors = (RTL_BITMAP
32714|                        | *)MemAllocatePoolWithTag(PagedPool,Count+sizeof(RTL_BITMAP
32715|                        | AP),PSMSECTORBITTAG);
32716|                        if ( !p->PSMSectors ) {
32717|                            // out of memory, dang, abort..
32718|                            Status =
32719|                            | STATUS_INSUFFICIENT_RESOURCES;
32720|                            DoneWithSnapshot(p); // don't
32721|                            | leave reference count too high
32722|                            break;
32723|                        }

```

```

32715|
32716|         Debug(DEBUG_DCPSM,("Dcpsm:
| Exclusive: Allocated %d bytes for volume
| '%S'\n",Count,DevExt->Name));
32717|
32718|         | RtlInitializeBitMap(p->PSMSectors,(PULONG)((char*)(p->PS
| MSectors)+sizeof(RTL_BITMAP)),Count*NUMBER_OF_BITS_IN_A_
| BYTE);
32719|         // set to PSM everything
32720|         RtlSetAllBits(p->PSMSectors);
32721|         MemCheckPool(p->PSMSectors);
32722|     }
32723| }
32724|
| p=GetNextSnapShotForMaster(&Master->SnapShots,p);
32725| }
32726| } __finally {
32727|     ReleaseSnapShotForWrite();
32728| }
32729|
32730| // free memory since we aborted with error.
32731| if ( !NT_SUCCESS(Status) ) {
32732|     DelnitForExclusive(User,Master);
32733| }
32734| #endif
32735| return Status;
32736| }
32737|
32738| /*-----
| -----*/
32739| NTSTATUS SbOpenExclusive( pkSnapShotMaster SnapShot )
32740| {
32741|     pOT_USER User=NULL;
32742|     NTSTATUS Status=STATUS_SUCCESS;
32743|
32744|     PAGED_CODE();
32745|
32746|     if ( SnapShot==NULL ) {
32747|         // we do not allow exclusive to be called with
| a null pointer.
32748|         Debug(DEBUG_INFO,("PSM Open Exclusive Called
| with NULL\n",PsGetCurrentProcess() ,
| PsGetCurrentThread()));
32749|         return STATUS_INVALID_PARAMETER;
32750|     }
32751|
32752| #if 1
32753| // TESTTEST
32754|     Debug(DEBUG_INFO,("PSM Open Exclusive Called %08x

```

```

    | %08x\n",PsGetCurrentProcess() , PsGetCurrentThread()));
32755|
32756|  /*lint -save -e740 */
32757|  User = FindPSMUser( PsGetCurrentProcess(),
    | PsGetCurrentThread() );
32758|  /*lint -restore */
32759|  if ( !User ) {
32760|      Debug(DEBUG_DCPSM,("SbOpenExclusive: PSM can
    | not find user!!!! failing open %08x,
    | %08x\n",PsGetCurrentProcess(), PsGetCurrentThread()));
32761|      return STATUS_INVALID_HANDLE;
32762|  }
32763|
32764|  if ( !User->Open ) {
32765|      Debug(DEBUG_DCPSM,("SbOpenExclusive: User does
    | not have psm open!"));
32766|      return STATUS_INVALID_HANDLE;
32767|  }
32768|
32769|  if ( SnapShot->ExclusiveProcess!=0 ) {
32770|      Debug(DEBUG_DCPSM,("SbOpenExclusive: Someone
    | else (%08x) already has exclusive
    | access!",SnapShot->ExclusiveProcess));
32771|      return PSM_ERROR_LOCKED_EXCLUSIVE;
32772|  }
32773|
32774| #if 0
32775|  // doesnt make sense any more for multiple snapshots
    | as long as the above test passed
32776|
32777|  // see if any other processes have psm open as we
    | cant get exclusive if someone else has us open
32778|  // note: another thread in the same process can
    | have opened psm, which is why we do not check the
32779|  // the global psm open flag. We allow this as we
    | assume the "process" knows what it is doing.
32780|
32781|  pmAcquireMutex ( &PSMUserMutex, NULL );
32782|  Next=GlobalData->PSMUsers;
32783|  while ( Next ) {
32784|      if ( (Next->ProcessID != PsGetCurrentProcess())
    | && (Next->Open) ) {
32785|          pmReleaseMutex ( &PSMUserMutex );
32786|          return PSM_ERROR_LOCKED_EXCLUSIVE;
32787|      }
32788|      Next = Next->Next;
32789|  }
32790|  pmReleaseMutex ( &PSMUserMutex );
32791| #endif
32792|

```



```

32793|   if ( AcquireOpenCloseResource()==STATUS_WAIT_0 ) {
32794|
32795|       __try {
32796|           // make sure someone else didnt get it
32797|           | first..
32797|           if ( SnapShot->ExclusiveProcess!=0 ) {
32798|               Debug(DEBUG_DCPSM,("SbOpenExclusive:
32799|               | Someone else (%08x) already has exclusive
32800|               | access!",SnapShot->ExclusiveProcess));
32799|               try_return (Status =
32801|               | PSM_ERROR_LOCKED_EXCLUSIVE);
32801|           }
32802|           Status = InitForExclusive(User,SnapShot);
32803|           if ( NT_SUCCESS(Status) ) {
32804|               // we can grant exclusive access!
32805|               SnapShot->ExclusiveProcess =
32806|               | PsGetCurrentProcess();
32806|           } else {
32807|               Debug(DEBUG_DCPSM,("Dcpsm:
32808|               | OpenExclusive: error %08x initing
32809|               | exclusive\n",Status));
32808|           }
32809|           try_exit: NOTHING;
32810|       } __finally {
32811|           ReleaseOpenCloseResource();
32812|       }
32813|   } else {
32814|       Status = PSM_CANCELED_BY_USER;
32815|   }
32816| #endif
32817|   return Status;
32818| }
32819|
32820| /*-----
32821|   | -----*/
32821| NTSTATUS SbCloseExclusive( pkSnapShotMaster SnapShot )
32822| {
32823|   pOT_USER   User=NULL;
32824|
32825|   PAGED_CODE();
32826|
32827|   if ( SnapShot==NULL ) {
32828|       // we do not allow exclusive to be called with
32829|       | a null pointer.
32829|       Debug(DEBUG_INFO,("PSM Close Exclusive Called
32830|       | with NULL\n",PsGetCurrentProcess() ,
32831|       | PsGetCurrentThread()));
32830|       return STATUS_INVALID_PARAMETER;
32831|   }

```

```

32832|
32833| #if 1
32834| // TESTTEST
32835|   Debug(DEBUG_INFO,("PSM Close Exclusive Called %08x
    | %08x\n",PsGetCurrentProcess() , PsGetCurrentThread()));
32836|
32837|   /*lint -save -e740 */
32838|   User = FindPSMUser( PsGetCurrentProcess(),
    | PsGetCurrentThread() );
32839|   /*lint -restore */
32840|   if ( !User ) {
32841|       Debug(DEBUG_DCPSM,("SbCloseExclusive: PSM can
    | not find user!!!! failing close %08x,
    | %08x\n",PsGetCurrentProcess(), PsGetCurrentThread()));
32842|       return STATUS_INVALID_HANDLE;
32843|   }
32844|
32845|   if ( !User->Open ) {
32846|       Debug(DEBUG_DCPSM,("SbCloseExclusive: User does
    | not have psm open!"));
32847|       return STATUS_INVALID_HANDLE;
32848|   }
32849|
32850|   // cant free if they didnt call..
32851|   if (
    | SnapShot->ExclusiveProcess!=PsGetCurrentProcess() ) {
32852|       Debug(DEBUG_DCPSM,("SbCloseExclusive: Someone
    | else (%08x) already has exclusive
    | access!",SnapShot->ExclusiveProcess));
32853|       return PSM_ERROR_LOCKED_EXCLUSIVE;
32854|   }
32855|
32856|   if (
    | AcquireOpenCloseResourceOnly(NULL)==STATUS_WAIT_0 ) {
32857|
32858|       __try {
32859|           DelInitForExclusive(User,SnapShot);
32860|       } __finally {
32861|           SnapShot->ExclusiveProcess = 0;
32862|           ReleaseOpenCloseResource();
32863|       }
32864|   } else {
32865|       return PSM_CANCELED_BY_USER;
32866|   }
32867| #endif
32868|   return STATUS_SUCCESS;
32869| }
32870|
32871|
32872| ULONG NumberOfSnapShotsInGroup (

```

```

32873|          ULONG
    | GroupNumber,
32874|          PDEVICE_OBJECT DevObj )
32875| {
32876|     ULONG          NumInGroup = 0;
32877|     PFILTERED_EXTENSION DevExt = NULL;
32878|     pkSnapshotEntry    p = NULL;
32879|
32880|     __try {
32881|         if ( DevObj != NULL ) {
32882|             if (
    | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
32883|                 DevExt = GetFilteredExtension(DevObj);
32884|                 __try {
32885|                     GetSnapshotForRead();
32886|                     __try {
32887|                         p =
    | GetTopSnapshot(&DevExt->Snapshots);
32888|                         while ( p ) {
32889|                             if ( GroupNumber ==
    | p->MasterSnapshot->GroupNumber ) {
32890|                                 NumInGroup++;
32891|                             }
32892|
    | p=GetNextSnapshot(&DevExt->Snapshots,p);
32893|                         }
32894|                     } __finally {
32895|                         ReleaseSnapshotForRead();
32896|                     }
32897|                 }
    | __except(ExceptionFilter(GetExceptionInformation())) {
32898|
    | Debug(DEBUG_DCPDM,("NumberOfSnapshotsInGroup: Exception
    | %08x for device %08x\n",GetExceptionCode(),DevObj));
32899|             }
32900|         }
32901|     }
32902| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
32903|     Debug(DEBUG_DCPDM,("NumberOfSnapshotsInGroup:
    | Exception %08x\n",GetExceptionCode()));
32904| }
32905|
32906| return NumInGroup;
32907| }
32908|
32909| ULONG NumberOfSnapshotsForVolume (PDEVICE_OBJECT
    | DevObj)
32910| {
32911|     ULONG          Num = 0;

```

```

32912|  PFILTERED_EXTENSION  DevExt = NULL;
32913|  pkSnapshotEntry      p = NULL;
32914|
32915|  __try {
32916|      if ( DevObj != NULL ) {
32917|          if (
32918|              | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
32919|                  DevExt = GetFilteredExtension(DevObj);
32920|                  __try {
32921|                      GetSnapshotForRead();
32922|                      __try {
32923|                          p =
32924|                          | GetTopSnapshot(&DevExt->Snapshots);
32925|                          while ( p ) {
32926|                              Num++;
32927|                              | p=GetNextSnapshot(&DevExt->Snapshots,p);
32928|                              }
32929|                              } __finally {
32930|                                  ReleaseSnapshotForRead();
32931|                              }
32932|                              | __except(ExceptionFilter(GetExceptionInformation())) {
32933|                                  Debug(DEBUG_DCPSPM,("NumberOfSnapshotsInGroup: Exception
32934|                                  | %08x for device %08x\n",GetExceptionCode(),DevObj));
32935|                                  }
32936|                                  }
32937|                                  | __except(ExceptionFilter(GetExceptionInformation())) {
32938|                                      Debug(DEBUG_DCPSPM,("NumberOfSnapshotsInGroup:
32939|                                      | Exception %08x\n",GetExceptionCode()));
32940|                                      }
32941|                                      }
32942|                                      return Num;
32943|                                  }
32944|                                  void DeleteOldestSnapshotInGroup( pOT_USER User,
32945|                                  | pkSnapshotMaster Master )
32946|                                  {
32947|                                      pkSnapshotMaster Oldest=NULL;
32948|                                      ULONG NumInGroup=0;
32949|                                      PDEVICE_OBJECT  DevObj=NULL;
32950|                                      PFILTERED_EXTENSION DevExt=NULL;
32951|                                      pkSnapshotEntry p;
32952|                                      if ( Master->NumToKeep==1 ) {
32953|                                          // nothing to do
32954|                                          return;

```

```

32953|    }
32954|
32955|    /*
32956|        This function works by looking at the
32957|        | snapshots attached to one volume as opposed
32958|        | to looking at all snapshots in the system.
32959|        | We do this as this will guarantee that only
32960|        | one master snapshot per snapshot entry is
32961|        | in the list, so we do not need to keep track
32962|        | of the fact that we already have counted
32963|        | that snapshot.
32964|
32965|        This assumes that ALL snapshots for this
32966|        | group cover the same volumes. For example,
32967|        | If C and D are snapshotted in group 5, then
32968|        | C and D will always be there for group 5. If
32969|        | however C and D are applied to group 5,
32970|        | then only C, then only D, this function will fail (and
32971|        | find only 2 snapshots for group 5, not 3 as
32972|        | it should be). If this is needed, this function
32973|        | needs to be adjusted to take into account
32974|        | that NumInGroup is incremented based on the number
32975|        | of snapshots found not the number of
32976|        | MasterSnapShots found.
32977|    */
32978|
32979|    // start at the top
32980|    DevObj = NULL;
32981|
32982|    __try {
32983|        // get a volume object that this snapshot
32984|        | applies to
32985|        GetSnapShotForRead();
32986|        __try {
32987|
32988|            | p=GetTopSnapShotForMaster(&Master->SnapShots);
32989|            DevObj = p->DeviceObject;
32990|            DoneWithSnapShot(p);
32991|        } __finally {
32992|            ReleaseSnapShotForRead();
32993|        }
32994|
32995|        // now go through all snapshots for this volume
32996|        | and count
32997|        // the snapshots along with finding the oldest
32998|        if ( DevObj != NULL ) {
32999|            if (
33000|            | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
33001|                DevExt = GetFilteredExtension(DevObj);
33002|                DoAgain:

```

```

32989|         __try {
32990|             GetSnapShotForRead();
32991|             __try {
32992|
32993|                 | p=GetTopSnapShot(&DevExt->SnapShots);
32994|                 while ( p ) {
32995|                     if ( Master->GroupNumber ==
32996|                         | p->MasterSnapShot->GroupNumber ) {
32997|                         // dont count always
32998|                         | keeps against group
32999|                         if (
33000|                             | p->MasterSnapShot->Priority!=255 ) {
33001|                                 // count self in
33002|                                 | this number
33003|                                 NumInGroup++;
33004|
33005|                                 if (
33006|                                     | p->MasterSnapShot!=Master ) {
33007|                                         // not looking
33008|                                         | at self
33009|                                         if ( Oldest ) {
33010|                                             if (
33011|                                                 | p->MasterSnapShot->Priority<Oldest->Priority ) {
33012|                                                     // at a
33013|                                                     | lower priority so grab it, regardless of time
33014|                                                     Oldest
33015|                                                     | = p->MasterSnapShot;
33016|                                                     } else
33017|                                                     if (
33018|                                                         | p->MasterSnapShot->Priority==Oldest->Priority ) {
33019|                                                             // same
33020|                                                             | priority, compare times
33021|                                                             if (
33022|                                                                 | p->MasterSnapShot->SnapShotTime.QuadPart<Oldest->SnapShotTime.QuadPart ) {
33023|                                                                     //
33024|                                                                     | mark this one as the oldest and lowest priority found
33025|                                                                     | so far
33026|                                                                     Oldest = p->MasterSnapShot;
33027|                                                                 }
33028|                                                             }
33029|                                                         } else {
33030|                                                             // first
33031|                                                             | suitable snapshot found
33032|                                                             Oldest = p->MasterSnapShot;
33033|                                                         }
33034|                                                     }
33035|                                                 }
33036|                                             }
33037|                                         }
33038|                                     }
33039|                                 }
33040|                             }
33041|                         }
33042|                     }
33043|                 }
33044|             }
33045|         }
33046|     }
33047| }
33048|
33049| }
33050|
33051| }
33052|
33053| }
33054|
33055| }
33056|
33057| }
33058|
33059| }
33060|
33061| }
33062|
33063| }
33064|
33065| }
33066|
33067| }
33068|
33069| }
33070|
33071| }
33072|
33073| }
33074|
33075| }
33076|
33077| }
33078|
33079| }
33080|
33081| }
33082|
33083| }
33084|
33085| }
33086|
33087| }
33088|
33089| }
33090|
33091| }
33092|
33093| }
33094|
33095| }
33096|
33097| }
33098|
33099| }
33100|
33101| }
33102|
33103| }
33104|
33105| }
33106|
33107| }
33108|
33109| }
33110|
33111| }
33112|
33113| }
33114|
33115| }
33116|
33117| }
33118|
33119| }
33120|
33121| }
33122|
33123| }
33124|
33125| }
33126|
33127| }
33128|
33129| }
33130|
33131| }
33132|
33133| }
33134|
33135| }
33136|
33137| }
33138|
33139| }
33140|
33141| }
33142|
33143| }
33144|
33145| }
33146|
33147| }
33148|
33149| }
33150|
33151| }
33152|
33153| }
33154|
33155| }
33156|
33157| }
33158|
33159| }
33160|
33161| }
33162|
33163| }
33164|
33165| }
33166|
33167| }
33168|
33169| }
33170|
33171| }
33172|
33173| }
33174|
33175| }
33176|
33177| }
33178|
33179| }
33180|
33181| }
33182|
33183| }
33184|
33185| }
33186|
33187| }
33188|
33189| }
33190|
33191| }
33192|
33193| }
33194|
33195| }
33196|
33197| }
33198|
33199| }
33200|
33201| }
33202|
33203| }
33204|
33205| }
33206|
33207| }
33208|
33209| }
33210|
33211| }
33212|
33213| }
33214|
33215| }
33216|
33217| }
33218|
33219| }
33220|
33221| }
33222|
33223| }
33224|
33225| }
33226|
33227| }
33228|
33229| }
33230|
33231| }
33232|
33233| }
33234|
33235| }
33236|
33237| }
33238|
33239| }
33240|
33241| }
33242|
33243| }
33244|
33245| }
33246|
33247| }
33248|
33249| }
33250|
33251| }
33252|
33253| }
33254|
33255| }
33256|
33257| }
33258|
33259| }
33260|
33261| }
33262|
33263| }
33264|
33265| }
33266|
33267| }
33268|
33269| }
33270|
33271| }
33272|
33273| }
33274|
33275| }
33276|
33277| }
33278|
33279| }
33280|
33281| }
33282|
33283| }
33284|
33285| }
33286|
33287| }
33288|
33289| }
33290|
33291| }
33292|
33293| }
33294|
33295| }
33296|
33297| }
33298|
33299| }
33300|
33301| }
33302|
33303| }
33304|
33305| }
33306|
33307| }
33308|
33309| }
33310|
33311| }
33312|
33313| }
33314|
33315| }
33316|
33317| }
33318|
33319| }
33320|
33321| }
33322|
33323| }
33324|
33325| }
33326|
33327| }
33328|
33329| }
33330|
33331| }
33332|
33333| }
33334|
33335| }
33336|
33337| }
33338|
33339| }
33340|
33341| }
33342|
33343| }
33344|
33345| }
33346|
33347| }
33348|
33349| }
33350|
33351| }
33352|
33353| }
33354|
33355| }
33356|
33357| }
33358|
33359| }
33360|
33361| }
33362|
33363| }
33364|
33365| }
33366|
33367| }
33368|
33369| }
33370|
33371| }
33372|
33373| }
33374|
33375| }
33376|
33377| }
33378|
33379| }
33380|
33381| }
33382|
33383| }
33384|
33385| }
33386|
33387| }
33388|
33389| }
33390|
33391| }
33392|
33393| }
33394|
33395| }
33396|
33397| }
33398|
33399| }
33400|
33401| }
33402|
33403| }
33404|
33405| }
33406|
33407| }
33408|
33409| }
33410|
33411| }
33412|
33413| }
33414|
33415| }
33416|
33417| }
33418|
33419| }
33420|
33421| }
33422|
33423| }
33424|
33425| }
33426|
33427| }
33428|
33429| }
33430|
33431| }
33432|
33433| }
33434|
33435| }
33436|
33437| }
33438|
33439| }
33440|
33441| }
33442|
33443| }
33444|
33445| }
33446|
33447| }
33448|
33449| }
33450|
33451| }
33452|
33453| }
33454|
33455| }
33456|
33457| }
33458|
33459| }
33460|
33461| }
33462|
33463| }
33464|
33465| }
33466|
33467| }
33468|
33469| }
33470|
33471| }
33472|
33473| }
33474|
33475| }
33476|
33477| }
33478|
33479| }
33480|
33481| }
33482|
33483| }
33484|
33485| }
33486|
33487| }
33488|
33489| }
33490|
3
```

```

33020|         }
33021|         | p=GetNextSnapShot(&DevExt->SnapShots,p);
33022|         }
33023|         } __finally {
33024|             ReleaseSnapShotForRead();
33025|         }
33026|     }
    | __except(ExceptionFilter(GetExceptionInformation())) {
33027|         | Debug(DEBUG_DCPSM,("DeleteOldestSnapShotInGroup:
    | Exception %08x for device
    | %08x\n",GetExceptionCode(),DevObj));
33028|     }
33029| }
33030| }
33031|
33032| // only delete if greater than number to keep
33033| if ( NumInGroup>Master->NumToKeep ) {
33034|     if ( Oldest ) {
33035|         | Debug(DEBUG_DCPSM,("DeleteOldestSnapShotInGroup:
    | Deleting Snapshot %08x\n",Oldest));
33036|         InternalClosePSM(User,Oldest);
33037|         // loop deleting until number of
    | snapshots in group is below how many to keep
33038|         NumInGroup=0;
33039|         Oldest=NULL;
33040|         goto DoAgain;
33041|     } else {
33042|         | Debug(DEBUG_DCPSM,("DeleteOldestSnapShotInGroup: Oldest
    | is null, most likely, new snapshot is only one\n"));
33043|     }
33044| } else {
33045|     | Debug(DEBUG_DCPSM,("DeleteOldestSnapShotInGroup: Only
    | %d snapshots in group, we need
    | %d\n",NumInGroup,Master->NumToKeep));
33046| }
33047| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
33048|         | Debug(DEBUG_DCPSM,("DeleteOldestSnapShotInGroup:
    | Exception %08x deleting snapshot
    | %08x\n",GetExceptionCode(),Oldest));
33049|     }
33050|
33051| return;
33052| }

```

```

33053|
33054| void LogOpen( tOpenTransactionInInternal *Buffer,
    | pkSnapshotMaster MasterSnapshot )
33055| {
33056|     __try {
33057|         if ( gLogOpenClose ) {
33058|             ULONG *DumpData;
33059|             WCHAR ErrorStr[12];
33060|             WCHAR *Strings[1];
33061|             swprintf(ErrorStr,L"%08x",MasterSnapshot);
33062|             Strings[0] = ErrorStr;
33063| #define NumDumpItems 10
33064|             ULONG
    | SizeOfDumpData=(NumDumpItems*sizeof(ULONG));
33065|
33066|             DumpData = (ULONG *)
    | MemAllocatePoolWithTag(PagedPool,SizeOfDumpData,TEMPTAG)
    | ;
33067|
33068|             if ( DumpData ) {
33069|
33070|                 DumpData[0] = (ULONG)MasterSnapshot;
33071|                 DumpData[1] =
    | Buffer->SizeOfCacheFileMB;
33072|                 DumpData[2] =
    | Buffer->MaxSizeOfCacheFileMB;
33073|                 DumpData[3] = Buffer->Flags;
33074|                 DumpData[4] = Buffer->QuiescentWait;
33075|                 DumpData[5] = Buffer->QuiescentTimeout;
33076|                 DumpData[6] = MasterSnapshot->Instance;
33077|                 DumpData[7] =
    | MasterSnapshot->SnapshotTime.LowPart;
33078|                 DumpData[8] =
    | MasterSnapshot->SnapshotTime.HighPart;
33079|                 DumpData[9] = Buffer->NumberOfDevices;
33080|
33081|             } else {
33082|                 SizeOfDumpData = 0;
33083|             }
33084| #undef NumDumpItems
33085|
33086|             /*lint -save -e740 */
33087|
    | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_OPEN
    | ED_INFORMATION,0,DumpData,SizeOfDumpData,Strings,1);
33088|             /*lint -restore */
33089|             if ( DumpData ) {
33090|                 FREE_POINTER(DumpData);
33091|             }
33092|         }

```



```

33093|    }
33094|    | __except(ExceptionFilter(GetExceptionInformation())) {
33095|        Debug(DEBUG_DCPSM,("LogOpen: Exception
33096|        | %08x\n",GetExceptionCode()));
33097|    }
33098| }
33099| NTSTATUS GetWin32NameFromNTName( WCHAR *NTName, WCHAR
33100|    | *DriveString, ULONG BufferSize )
33101| {
33102|    // FIXFIXFIX we need to code this function
33103|    return STATUS_NOT_FOUND;
33104| }
33105| NTSTATUS GetNTNameFromWin32Name( WCHAR *Win32Name,
33106|    | WCHAR *NTName )
33107| {
33108|    UNICODE_STRING  UniName={0};
33109|    OBJECT_ATTRIBUTES ObjectAttributes={0};
33110|    NTSTATUS        Status=STATUS_SUCCESS;
33111|    HANDLE          SymHandle=NULL;
33112|    WCHAR           Name[256];
33113|    wcscpy(Name,Win32Name);
33114|    Debug(DEBUG_DCPSM,("GetNTNameFromWin32Name: Looking
33115|    | for '%S'\n",Name));
33116|    RtlInitUnicodeString( &UniName, Name);
33117|    InitializeObjectAttributes ( &ObjectAttributes,
33118|        &UniName,
33119|        OBJ_CASE_INSENSITIVE,
33120|        NULL,
33121|        NULL );
33122|    Status = ZwOpenSymbolicLinkObject( &SymHandle,
33123|        | STANDARD_RIGHTS_READ, &ObjectAttributes);
33124|    if ( NT_SUCCESS(Status) ) {
33125|        ULONG Ret=0;
33126|        UniName.Length = 0;
33127|        UniName.MaximumLength = 256;
33128|        Status =
33129|        | ZwQuerySymbolicLinkObject(SymHandle,&UniName,&Ret);
33130|        if ( NT_SUCCESS(Status) ) {
33131|            Debug(DEBUG_DCPSM,("GetNTNameFromWin32Name:
33132|            | Device Name = '%wZ'\n",&UniName));
33133|            wcscpy(NTName,UniName.Buffer);
33134|        } else {

```

```

33135|         Debug(DEBUG_DCPSM,("GetNTNameFromWin32Name:
| Error %08x opening symlink '%S'\n",Status,Name));
33136|     }
33137|
33138|     ZwClose(SymHandle);
33139|     SymHandle = NULL;
33140| } else {
33141|     Debug(DEBUG_DCPSM,("GetNTNameFromWin32Name:
| Error %08x\n",Status));
33142| }
33143| return Status;
33144| }
33145|
33146| NTSTATUS GetDriveLetterFromNTName( WCHAR *NTName, WCHAR
| *DriveString, ULONG BufferSize )
33147| {
33148|     WCHAR      DriveName[20] = L"\\DosDevices\\C:";
33149|     UNICODE_STRING UniName={0};
33150|     OBJECT_ATTRIBUTES ObjAttr={0};
33151|     HANDLE      SymHandle=NULL;
33152|     WCHAR      SymSpace[256]={0};
33153|     NTSTATUS     Status=STATUS_UNSUCCESSFUL;
33154|     ULONG      Ret=0;
33155|     UCHAR      DriveLetter=0;
33156|     NTSTATUS RetStatus = STATUS_NOT_FOUND;
33157|
33158|     PAGED_CODE();
33159|
33160|     for ( DriveLetter=2;DriveLetter<26;DriveLetter++ )
| {
33161|
33162|         DriveName[12] = DriveLetter+65;
33163|
33164|         RtlInitUnicodeString( &UniName, DriveName );
33165|
33166|         ObjAttr.Length          =
| sizeof(ObjAttr);
33167|         ObjAttr.RootDirectory    = NULL;
33168|         ObjAttr.Attributes      =
| OBJ_CASE_INSENSITIVE;
33169|         ObjAttr.ObjectName       = &UniName;
33170|         ObjAttr.SecurityDescriptor = NULL;
33171|         ObjAttr.SecurityQualityOfService = NULL;
33172|
33173|         Status = ZwOpenSymbolicLinkObject( &SymHandle,
| STANDARD_RIGHTS_READ, &ObjAttr );
33174|         if ( NT_SUCCESS(Status) ) {
33175|             UniName.Length = 0;
33176|             UniName.MaximumLength = 256;
33177|             UniName.Buffer = SymSpace;

```

```

33178|         Status =
33179|         | ZwQuerySymbolicLinkObject(SymHandle,&UniName,&Ret);
33180|         if ( NT_SUCCESS(Status) ) {
33181|             Debug(DEBUG_DCPSM,("Symlink '%S' ==
33182|             | '%wZ\\n",DriveName,&UniName));
33183|             if (
33184|                 | _wcsnicmp(UniName.Buffer,NTName,wcslen(NTName))==0 ) {
33185|                 // \\DosDevices\C:
33186|                 // 0123456789-12
33187|                 DriveString[0] = DriveLetter+65;
33188|                 DriveString[1] = 0;
33189|                 ZwClose(SymHandle);
33190|                 RetStatus= STATUS_SUCCESS;
33191|                 break;
33192|             }
33193|         } else {
33194|             Debug(DEBUG_DCPSM,("Error
33195|             | %08x\\n",Status));
33196|         }
33197|         ZwClose(SymHandle);
33198|     } else {
33199|         //Debug(DEBUG_DCPSM,("Error
33200|         | %08x\\n",Status));
33201|     }
33202| } // for
33203|
33204| return RetStatus;
33205| }
33206|
33207| NTSTATUS MakeVolumeListString(
33208|     pkSnapShotMaster Master,
33209|     WCHAR *String,
33210|     ULONG StringSize
33211| )
33212| {
33213|     NTSTATUS Status=STATUS_SUCCESS;
33214|     __try {
33215|         wcscpy(String,L "");
33216|         Status = ValidateKernelSnapShotPointer(Master);
33217|         if ( NT_SUCCESS(Status) ) {
33218|             WCHAR *Buffer;
33219|             ULONG BufferSize=128*1024;
33220|
33221|             Buffer =
33222|             | (WCHAR*)MemAllocatePoolWithTag(PagedPool,BufferSize,TEMP

```

```

    | TAG);
33222|         if ( Buffer ) {
33223|             Status = SbGetKernelSnapShotVolumes(
33224|
    | Master,
33225|
    | Buffer,
33226|
    | &BufferSize);
33227|
33228|         if ( !Status ) {
33229|             WCHAR *p=Buffer;
33230|             WCHAR Temp[512];
33231|
33232|             while ( *p ) {
33233|                 // use drive letter
33234|                 if (
    | GetDriveLetterFromNTName(p,Temp,sizeof(Temp))!=STATUS_SU
    | CCESS ) {
33235|                     // no drive letter, so lets
    | just mark the spot
33236|                     // as the packet can not
    | get bigger than 150 bytes.
33237|                     wcscpy(Temp,L"*");
33238| /*
33239|
    | // use win 32 name
33240|
    | if(GetWin32NameFromNTName(p,Temp,sizeof(Temp))!=STATUS_S
    | UCESS) {
33241|
    | // use nt name
33242|
    | wcscpy(Temp,p);
33243|
    | }
33244| */
33245|         }
33246|         if ( NumBytes(Temp)>=StringSize
    | ) {
33247|             Status =
    | STATUS_BUFFER_TOO_SMALL;
33248|             break;
33249|         } else {
33250|             wcscat(String,Temp);
33251|             StringSize-=NumBytes(Temp);
33252|         }
33253|
33254|         p+=wcslen(p)+1;
33255|

```

```

33256|         if ( *p ) {
33257|             // add comma if another
33258|             | volume
33258|             StringSize-=2;
33259|             wscat(String,L,"");
33260|         }
33261|     }
33262| }
33263|     FREE_POINTER(Buffer);
33264|     Status = STATUS_SUCCESS;
33265| } else {
33266|
33267|     | Debug(DEBUG_DCPSM,("MakeVolumeListString: Out of
33268|     | memory\n"));
33267|     Status = STATUS_INSUFFICIENT_RESOURCES;
33268|     }
33269| } else {
33270|     Debug(DEBUG_DCPSM,("MakeVolumeListString:
33271|     | invalid master snapshot %08x\n",Master));
33271|     }
33272| }
33273|     | __except(ExceptionFilter(GetExceptionInformation())) {
33273|     Status = GetExceptionCode();
33274|     Debug(DEBUG_DCPSM,("RemoveUserLinks: Exception
33275|     | %08x\n",Status));
33275|     }
33276|     return Status;
33277| }
33278|
33279|
33280| void LogLinkCreated( pkSnapShotMaster MasterSnapShot )
33281| {
33282|     __try {
33283|         if ( gLogOpenClose ) {
33284|             WCHAR *Strings[9];
33285|             WCHAR VolList[100];
33286|             WCHAR ID[10];
33287|             WCHAR Month[4];
33288|             WCHAR Day[4];
33289|             WCHAR Year[6];
33290|             WCHAR Hour[4];
33291|             WCHAR Minute[4];
33292|             WCHAR Second[4];
33293|             TIME_FIELDS tm;
33294|             LARGE_INTEGER Local;
33295|             ULONG DumpData;
33296|
33297|             wcscpy(VolList,L "");
33298|
33299|             | MakeVolumeListString(MasterSnapShot,VolList,sizeof(VolLi

```

```

    | st));
33299|         swprintf(ID,L"%08x",MasterSnapShot);
33300|
33301|
    | ExSystemTimeToLocalTime(&MasterSnapShot->SnapShotTime,&L
    | ocal);
33302|         RtlTimeToTimeFields(&Local,&tm);
33303|         swprintf(Month,L"%02d",tm.Month);
33304|         swprintf(Day,L"%02d",tm.Day);
33305|         swprintf(Year,L"%4d",tm.Year);
33306|         swprintf(Hour,L"%02d",tm.Hour);
33307|         swprintf(Minute,L"%02d",tm.Minute);
33308|         swprintf(Second,L"%02d",tm.Second);
33309|
33310|         wcscat(VolList,L":\\");
33311|
33312|         ULONG LogSize =
    | ((sizeof(IO_ERROR_LOG_MESSAGE)-sizeof(IO_ERROR_LOG_PACKE
    | T)) +
33313|             ERROR_LOG_MAXIMUM_SIZE) -
33314|
    | sizeof(IO_ERROR_LOG_MESSAGE) -
33315|             NumBytes(ID) -
33316|             NumBytes(Month) -
33317|             NumBytes(Day) -
33318|             NumBytes(Year) -
33319|             NumBytes(Hour) -
33320|             NumBytes(Minute) -
33321|             NumBytes(Second) -
33322|
    | sizeof(DumpData)-(sizeof(WCHAR) * 8);
33323|
33324|         DumpData = (ULONG)MasterSnapShot;
33325|
33326|         if ( NumBytes(VolList)>=LogSize ) {
33327|             // volume list is too big
33328|             wcscpy(VolList,L"...");
33329|         }
33330|
33331|         LogSize-=NumBytes(VolList);
33332|
33333|         if (
    | _wcsicmp(MasterSnapShot->UserSnapShotName,L"")==0 ) {
33334|             // just leave blank
33335|         } else {
33336|             if (
    | NumBytes(MasterSnapShot->UserSnapShotName)<LogSize ) {
33337|
    | wcscat(VolList,MasterSnapShot->UserSnapShotName);
33338|         } else {

```

```

33339|          WCHAR
33340|          | *p=wcsrchr(MasterSnapShot->UserSnapShotName,L"\\");
33341|          if ( p ) {
33342|              p++;
33343|          #define DOT_SIZE (unsigned)(3*sizeof(WCHAR))
33344|          if (
33345|              | (NumBytes(p)+DOT_SIZE)<LogSize ) {
33346|                  wcscat(VolList,L"...");
33347|                  wcscat(VolList,p);
33348|              } else {
33349|                  Trunc:
33350|                  wcscat(VolList,L"...");
33351|                  if ( LogSize>DOT_SIZE ) {
33352|                      p =
33353|                      | MasterSnapShot->UserSnapShotName +
33354|                      | NumBytes(MasterSnapShot->UserSnapShotName) - LogSize -
33355|                      | 1;
33356|                      | p+=(DOT_SIZE/sizeof(WCHAR));
33357|                      wcscat(VolList,p);
33358|                  }
33359|              } else {
33360|                  goto Trunc;
33361|              }
33362|          }
33363|          #define NumStrings 8
33364|          Strings[0] = ID;
33365|          | // 2
33366|          Strings[1] = VolList;
33367|          | // 3
33368|          Strings[2] = Month;
33369|          | // 4
33370|          Strings[3] = Day;
33371|          | // 5
33372|          Strings[4] = Year;
33373|          | // 6
33374|          Strings[5] = Hour;
33375|          | // 7
33376|          Strings[6] = Minute;
33377|          | // 8
33378|          Strings[7] = Second;
33379|          | // 9
33380|          /*lint -save -e740 */
33381|          | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_LINK

```

```

    | _CREATED,0,&DumpData,1,Strings,NumStrings);
33374|         /*lint -restore */
33375|     }
33376| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
33377|     Debug(DEBUG_DCPSM,("LogLinkCreated: Exception
    | %08x\n",GetExceptionCode()));
33378| }
33379| }
33380|
33381|
33382| extern LARGE_INTEGER TimeOfLastLogEntry;
33383|
33384| //-----
    | -----
    | --
33385|
33386| NTSTATUS GetDeviceExtensionList (
33387|     tOpenTransactionInInternal    *In,
33388|     PFILTERED_EXTENSION           *&DevExtArray,
33389|     ULONG                         &NumDevices )
33390| {
33391|     NTSTATUS status = STATUS_UNSUCCESSFUL;
33392|     NumDevices = 0;
33393|
33394|     __try {
33395|         // We are creating an array of pointers to
    | filtered extensions.
33396|         // The parameter 'DevExtArray' is therefore a
33397|         // reference to a pointer to a pointer to a
    | filtered extension.
33398|
33399|         DevExtArray = (PFILTERED_EXTENSION *)
    | MemAllocatePoolWithTag(
33400|             PagedPool,
33401|             In->NumberOfDevices *
    | sizeof(PFILTERED_EXTENSION),
33402|             TEMPTAG );
33403|
33404|         if ( DevExtArray ) {
33405|             for ( ULONG i=0; i < In->NumberOfDevices;
    | ++i ) {
33406|                 PDEVICE_OBJECT DevObj =
    | (PDEVICE_OBJECT) GetObjectFromName( (WCHAR
    | *)DN_MakePointer(In,In->DeviceName[i]) );
33407|                 if ( DevObj ) {
33408|                     DevExtArray[NumDevices++] =
    | GetFilteredExtension (DevObj);
33409|                 }
33410|             }

```



```

33411|
33412|     status = STATUS_SUCCESS;
33413| } else {
33414|     status = STATUS_INSUFFICIENT_RESOURCES;
33415| }
33416| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
33417|     status = GetExceptionCode();
33418|     Debug(DEBUG_DCPSM,("!!! Exception %08x in
    | GetDeviceExtensionList\n",status));
33419| }
33420|
33421| if ( !NT_SUCCESS(status) ) {
33422|     NumDevices = 0;
33423|     if ( DevExtArray ) {
33424|         FREE_POINTER(DevExtArray);
33425|     }
33426| }
33427|
33428| return status;
33429| }
33430|
33431| //-----
    | -----
    | --
33432| //  Given a list of volumes that we want to take a
    | snapshot of, the following function
33433| //  determines whether any of the volumes have not
    | completed Part2 (by checking file handles)
33434| //  but have snapshots. If this is the case, the
    | function returns 'false', indicating that
33435| //  we are between part1 and part2, and that snapshot
    | creation should be delayed a bit.
33436|
33437| STATIC bool ReadyForSnapShotCreation (
    | tOpenTransactionInternal *In )
33438| {
33439|     bool IsReady = true;
33440|     PFILTERED_EXTENSION *DevExtArray = NULL;
33441|     ULONG NumDevices = 0;
33442|
33443|     __try {
33444|         NTSTATUS GetListStatus = GetDeviceExtensionList
    | (In, DevExtArray, NumDevices);
33445|         ASSERT(NT_SUCCESS(GetListStatus));
33446|         if ( NT_SUCCESS(GetListStatus) ) {
33447|             ASSERT (DevExtArray != NULL);
33448|
33449|             if ( DevExtArray ) {
33450|                 for ( ULONG i=0; i<NumDevices; ++i ) {

```

```

33451|             bool AllHandlesOpen =
33452|
33453|             | IsValidHandle(DevExtArray[i]->Cache.CacheFile.FileHandle
33454|             | ) &&
33455|             | IsValidHandle(DevExtArray[i]->Cache.IndexFile.FileHandle
33456|             | ) &&
33457|             | IsValidHandle(DevExtArray[i]->Cache.HeaderFile.FileHandle
33458|             | e);
33459|
33460|             if ( !AllHandlesOpen &&
33461|             | DevExtArray[i]->PSMed ) {
33462|                 IsReady = false;
33463|                 break;
33464|             }
33465|         }
33466|
33467|         FREE_POINTER (DevExtArray);
33468|     }
33469| }
33470|
33471| }
33472|
33473| //-----
33474| | -----
33475| | --
33476|
33477| NTSTATUS SbOpenPSM (
33478|     pOT_USER          User,
33479|     tOpenTransactionInInternal *Buffer,
33480|     ULONG              OTOSize,
33481|     tOpenTransactionOutInternal *OutBuffer,
33482|     PKEVENT            AbortEvent )
33483| {
33484|     NTSTATUS Status = STATUS_SUCCESS;
33485|     NTSTATUS CountStatus = STATUS_SUCCESS;
33486|     ULONG NumActiveSnapShots = 0;
33487|
33488|     PAGED_CODE();
33489|
33490|     MemTrackPrintStats( "OpenPSM" );

```

```

33489|   MemShowUsage();
33490|
33491|   Debug(DEBUG_DCPSM,("Entering SbOpenPSM: User=%08x,
    | AbortEvent=%08x\n",User,AbortEvent));
33492|
33493|   // exiting event not yet initialized, so use this
    | function
33494|   // otherwise we corrupt memory and/or bsod
33495|   if ( AcquireOpenCloseResourceOnly(AbortEvent) !=
    | STATUS_WAIT_0 ) {
33496|       Debug(DEBUG_DCPSM,("SbOpenPSM: (1) canceled by
    | AbortEvent\n"));
33497|       return PSM_CANCELED_BY_USER;
33498|   }
33499|
33500|   bool OpenCloseAcquired = true;
33501|   __try {
33502|       bool ReadyForSnap = false;
33503|       while ( !ReadyForSnap ) {
33504|
    | //-----
    | -----
33505|           // It is possible for SbOpenPSM to be
    | called while we are in a revert.
33506|           // Revert will be holding the OpenClose
    | until it is finished.
33507|           // However, we really want to make sure
    | that part2 has run.
33508|           // Instead of just grabbing the resource
    | and running with it, we need to
33509|           // check to see if part2 has finished. If
    | not, release OpenClose,
33510|           // delay for a little, and loop back to
    | try again. The idea is that
33511|           // this gives part2 a chance to grab the
    | resource and complete.
33512|           // This (hopefully) resolves defect #279
    | 'snapshot during revert'.
33513|
    | //-----
    | -----
33514|           ReadyForSnap =
    | ReadyForSnapshotCreation(Buffer);
33515|           Debug(DEBUG_DCPSM,("SbOpenPSM:
    | ReadyForSnapshotCreation returned
    | %s\n",(ReadyForSnap?"TRUE":"FALSE")));
33516|           if ( !ReadyForSnap ) {
33517|               ReleaseOpenCloseResource();
33518|               OpenCloseAcquired = false;    // so
    | we know not to release OpenClose again if exception

```

```

| happens
33519|
33520|         const int SecondsToWait = 5;
33521|         LARGE_INTEGER TimeToWait;
33522|         TimeToWait.QuadPart =
| RELATIVE(SECONDS(SecondsToWait));
33523|         KeDelayExecutionThread
| ((KPROCESSOR_MODE)KernelMode, FALSE, &TimeToWait);
33524|
33525|         if (
| AcquireOpenCloseResourceOnly(AbortEvent) !=
| STATUS_WAIT_0 ) {
33526|             Debug(DEBUG_DCPSM,("SbOpenPSM: (2)
| canceled by AbortEvent\n"));
33527|             return PSM_CANCELED_BY_USER;
33528|         }
33529|
33530|         OpenCloseAcquired = true;
33531|     }
33532| }
33533|
33534| #ifdef DEBUG
33535|     Debug(DEBUG_DCPSM,("SbOpenPSM: CacheFileName
| = '%S'\n",Buffer->CacheFileName));
33536|     Debug(DEBUG_DCPSM,("
| SizeOfCacheFileInMB =
| %08x\n",Buffer->SizeOfCacheFileMB));
33537|     Debug(DEBUG_DCPSM,("
| MaxSizeOfCacheFileInMB =
| %08x\n",Buffer->MaxSizeOfCacheFileMB));
33538|     Debug(DEBUG_DCPSM,("      Flags
| = %08x\n",Buffer->Flags));
33539|     Debug(DEBUG_DCPSM,("      Q Wait
| = %08x\n",Buffer->QuiescentWait));
33540|     Debug(DEBUG_DCPSM,("      Q Time
| = %08x\n",Buffer->QuiescentTimeout));
33541|     Debug(DEBUG_DCPSM,("      Number Devices
| = %08x\n",Buffer->NumberOfDevices));
33542|
33543|     {
33544|         ULONG i;
33545|         for ( i=0;i<Buffer->NumberOfDevices;i++ ) {
33546|             Debug(DEBUG_DCPSM,("      [%02x]
| =
| '%S'\n",i,DN_MakePointer(Buffer,Buffer->DeviceName[i]))
| ;
33547|         }
33548|     }
33549| #endif
33550|

```

```

33551| #if 0
33552| // done when the snapshot is used by someone else not
    | during the open of a new one
33553|     if ( (GlobalData->ExclusiveProcess!=0) &&
33554|         | (GlobalData->ExclusiveProcess!=PsGetCurrentProcess()) )
    | {
33555|         Debug(DEBUG_DCPSM,("SbOpenPsm: Someone else
    | (%08x) already has exclusive
    | access!",GlobalData->ExclusiveProcess));
33556|         | try_return(Status=PSM_ERROR_LOCKED_EXCLUSIVE);
33557|     }
33558| #endif
33559|
33560|     // clear last error time so event logs are sent
33561|     // based on the new snapshot
33562|     TimeOfLastLogEntry.QuadPart = 0;
33563|
33564|     // clear event incase user didnt
33565|     if ( User->ErrorEvent ) {
33566|         pmClearEvent(User->ErrorEvent);
33567|     }
33568|
33569|     /*
33570|     //OTM fossil: user-specified cache files are
    | now always ignored.
33571|     if ( !(Buffer->InternalFlags &
    | PSM_IFLAG_PERSISTENT) ) {
33572|         Status =
    | SblsADirectory(Buffer->CacheFileName);
33573|         if ( (Status == STATUS_FILE_IS_A_DIRECTORY)
    | || (Status == STATUS_OBJECT_PATH_SYNTAX_BAD) ) {
33574|             try_return(NOTHING);
33575|         }
33576|     }
33577|     */
33578|
33579|     // call preinit
33580|     if ( !PSManPSMInited ) {
33581|         | UpdateGlobalStatus(PSM_RESOURCE_ACQUISITION);
33582|         Status = SbPreInit(User,Buffer,AbortEvent);
33583|     } else {
33584|         // check to see if we are in a state where
    | ALL instances need to exit.
33585|         if ( (GlobalData->NumActive) &&
    | (LastErrorStatus!=0) ) {
33586|             Status = LastErrorStatus;
33587|         }

```

```

33588|     }
33589|
33590|     if ( NT_SUCCESS(Status) ) {
33591|         BOOLEAN WouldCauseMoreSnapShots = FALSE;
33592|
33593|         if ( Buffer->NumToKeep == -1 ||
33594|             | Buffer->NumToKeep == 0 ) {
33595|             // The user is requesting no limit on
33596|             | how many to keep
33597|             // for this group. Therefore, no old
33598|             | snapshots would be deleted.
33599|             WouldCauseMoreSnapShots = TRUE;
33600|         } else {
33601|             // Now we need to figure out whether we
33602|             | are already at the
33603|             // group limit. If so, creating a new
33604|             | snapshot will automatically
33605|             // delete an old one, so there is no
33606|             | problem.
33607|
33608|             PDEVICE_OBJECT DevObj =
33609|             | (PDEVICE_OBJECT) GetObjectFromName(
33610|             | (WCHAR *)DN_MakePointer(Buffer,Buffer->DeviceName[0])
33611|             | );
33612|
33613|             if ( DevObj ) {
33614|                 ULONG numInGroup =
33615|                 | NumberOfSnapShotsInGroup (
33616|                 | (ULONG) Buffer->CallerPrivateUse,    // group number
33617|                 | DevObj );
33618|
33619|                 if ( numInGroup < Buffer->NumToKeep
33620|                     | ) {
33621|                     WouldCauseMoreSnapShots = TRUE;
33622|                 }
33623|             } else {
33624|                 Debug(DEBUG_DCPSM,("SbOpenPSM:
33625|                 | Could not find DevObj in snapshot limit logic\n"));
33626|                 WouldCauseMoreSnapShots = TRUE;
33627|             }
33628|         }
33629|
33630|         if ( WouldCauseMoreSnapShots ) {
33631|             WCHAR Str1[10] = {0};
33632|             WCHAR *Strings[] = {Str1};
33633|             const ULONG MaxSnapShots =
33634|             | PersistentDictionary::QueryMaxNumUserSnapShots();

```

```

33623|
33624|         Debug(DEBUG_DCPSM,("SbOpenPSM:
| NumActiveSnapshots=%d, MaxSnapShots=%d\n",
33625|         | GlobalData->NumActive,
33626|         MaxSnapShots));
33627|
33628|         if ( GlobalData->NumActive >=
| MaxSnapShots ) {
33629|             // We are at the limit of how many
| user snapshots may exist.
33630|             // Therefore, we need to determine
| whether creating this
33631|             // new snapshot will really
| increase the number of snapshots.
33632|             // The number won't actually
| increase if creating this snapshot
33633|             // results in deleting another one
| due to NumToKeep rules for a group.
33634|             Debug(DEBUG_DCPSM,("SbOpenPSM:
| Maximum number of snapshots reached - trying to delete
| oldest.\n"));
33635|             BOOLEAN OldestSnapShotWasDeleted =
| DeleteOldestSnapShot(NULL,FALSE);
33636|             if ( OldestSnapShotWasDeleted ) {
33637|                 Debug(DEBUG_DCPSM,("SbOpenPSM:
| Snapshot count threshold reached - oldest snapshot was
| deleted.\n"));
33638|                 LogError (
33639|                 | (PDEVICE_OBJECT)PSManDriverObject,
33640|                 NULL,
33641|                 | PSM_OLDEST_SNAPSHOT_DELETED,
33642|                 | PSM_OLDEST_SNAPSHOT_DELETED,
33643|                 NULL,
33644|                 0,
33645|                 NULL,
33646|                 0);
33647|             } else {
33648|                 swprintf ( Str1, L"%d",
| MaxSnapShots );
33649|                 Debug(DEBUG_DCPSM,("SbOpenPSM:
| Rejecting snapshot creation due to reaching
| limit.\n"));
33650|                 LogError (
33651|                 | (PDEVICE_OBJECT)PSManDriverObject,
33652|                 NULL,

```

```

33653|
33654|     | PSM_MAXIMUM_ALLOWED_SNAPSHOTS_REACHED,
33655|         NULL,
33656|         0,
33657|         Strings,
33658|         1);
33659|     try_return (Status =
33660|         | PSM_MAXIMUM_ALLOWED_SNAPSHOTS_REACHED);
33661|     }
33662|     } else {
33663|         const ULONG WarningThresholdPercent
33664|         | = 80;      //!!! FIXFIXFIX: get from registry
33665|         ULONG WarningThreshold =
33666|         | (MaxSnapShots * WarningThresholdPercent) / 100;
33667|         if ( GlobalData->NumActive >=
33668|         | WarningThreshold ) {
33669|             if ( MaxSnapShots > 0 ) {
33670|                 swprintf ( Str1, L"%d",
33671|                 | (100 * GlobalData->NumActive) / MaxSnapShots );
33672|             } else {
33673|                 wcsncpy ( Str1, L"?" );
33674|                 | // should never happen, but don't divide by zero!
33675|             }
33676|             Debug(DEBUG_DCPSM,("SbOpenPSM:
33677|             | Reached snapshot warning threshold: logging error\n"));
33678|             LogError (
33679|             | (PDEVICE_OBJECT)PSManDriverObject,
33680|             | NULL,
33681|             | PSM_SNAPSHOT_COUNT_THRESHOLD_REACHED,
33682|             | PSM_SNAPSHOT_COUNT_THRESHOLD_REACHED,
33683|             | NULL,
33684|             | 0,
33685|             | Strings,
33686|             | 1);
33687|         }
33688|     }
33689| }
33690| }
33691| }
33692| PersistentDictionary::SetInfo(Buffer);
33693| /*
33694| // OTM fossil: Always call SetInfo,
33695| | whether persistent or temporary snapshot.
33696| if ( Buffer->InternalFlags &
33697| | PSM_IFLAG_PERSISTENT ) {

```



```

33689|         PersistentDictionary::SetInfo(Buffer);
33690|     }
33691|     */
33692|
33693|     // try and get the window and turn on PSM
33694|     pkSnapShotMaster MasterSnapShot = NULL;
33695|     Status = WaitForQuiescentPeriod( User,
    | Buffer, &MasterSnapShot, AbortEvent );
33696|     if ( NT_SUCCESS(Status) ) {
33697|         ASSERT (MasterSnapShot != NULL);
33698|         InterlockedIncrement((PLONG)
    | &GlobalData->NumActive);
33699|         InterlockedIncrement((PLONG)
    | &User->Open);
33700|         Debug(DEBUG_DCPSM,("PSM Opened
    | snapshot=%08x; NumActive incremented to %08x,
    | open=%08x\n",
33701|             MasterSnapShot,
33702|             GlobalData->NumActive,
33703|             User->Open));
33704|
33705|         PsmActive = TRUE;
33706|         LogOpen(Buffer,MasterSnapShot);
33707|
33708|
    | UpdateGlobalStatus(PSM_MAPPING_IN_SNAPSHOTS);
33709|
33710|         // map drives and get instance number
33711|         Status =
    | VDiskMapInDrives(MasterSnapShot,Buffer,OTOSize,OutBuffer
    | );
33712|
33713|         if ( !NT_SUCCESS(Status) ) {
33714|             // failed, clean up..
33715|
    | Debug(DEBUG_DCPSM,("VdiskMapInDrives Failed %08x for
    | %08x %08x\n",Status,User,MasterSnapShot));
33716|
    | InternalClosePSM(User,MasterSnapShot);
33717|
33718|         } else {
33719|
33720|
    | DeleteOldestSnapShotInGroup(User,MasterSnapShot);
33721|
33722|         if (
    | OTOSize>=sizeof(tOpenTransactionOutInternal) ) {
33723|             // return back pointer to this
    | so they can pass it to use on close
33724|

```

```

    | OutBuffer->KernelSnapShotPointer = MasterSnapShot;
33725|         OutBuffer->SnapShotTime =
    | MasterSnapShot->SnapShotTime;
33726|         OutBuffer->Instance =
    | MasterSnapShot->Instance;
33727|
33728|         GetSnapShotForRead();
33729|         __try {
33730|             pkSnapShotEntry s =
    | GetTopSnapShotForMaster(&MasterSnapShot->SnapShots);
33731|             if ( s ) {
33732|
    | s->Dictionary->GetOutParams(OutBuffer->CacheFileName);
33733|                 DoneWithSnapShot(s);
33734|             }
33735|         } __finally {
33736|             ReleaseSnapShotForRead();
33737|         }
33738|     }
33739| }
33740| } else {
33741|     Debug(DEBUG_DCPSM,("Wait for Q period
    | failed %08x\n",Status));
33742|     if ( GlobalData->NumActive==0 ) {
33743|         // only call if we need to deinit
33744|         InterlockedIncrement((PLONG)
    | &GlobalData->NumActive);
33745|         Debug(DEBUG_DCPSM,("SbOpenPSM:
    | Incremented NumActive to %08x (expecting
    | InternalClosePSM to decrement), open=%08x\n",
33746|             GlobalData->NumActive,
33747|             User->Open));
33748|
33749|         InternalClosePSM(User,NULL);
33750|     }
33751| }
33752| }
33753| try_exit:
33754|
33755| if ( !NT_SUCCESS(Status) ) {
33756|     WCHAR MessageString1 [64] = {0};
33757|     WCHAR *MessageStrings[10] =
    | {MessageString1};
33758|
33759|     switch ( Status ) {
33760|         case PSM_ERROR_TIMEOUT: {
33761|             swprintf (MessageString1,
    | L"%d", Buffer->QuiescentTimeout / 60);
33762|             /*lint -save -e740 */
33763|

```

```

    | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
    | R_TIMEOUT,PSM_ERROR_TIMEOUT,NULL,0,MessageStrings,1);
33764|          /*lint -restore */
33765|          break;
33766|      }
33767|
33768|      case STATUS_INSUFFICIENT_RESOURCES:
33769|      case PSM_ERROR_OUT_OF_MEMORY : {
33770|          /*lint -save -e740 */
33771|
    | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
    | R_OUT_OF_MEMORY,PSM_ERROR_OUT_OF_MEMORY,NULL,0,NULL,0);
33772|          /*lint -restore */
33773|          break;
33774|      }
33775|      case STATUS_OBJECT_NAME_NOT_FOUND:
33776|      case PSM_ERROR_NO_SUCH_OBJECT: {
33777|          /*lint -save -e740 */
33778|
    | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
    | R_NO_SUCH_OBJECT,PSM_ERROR_NO_SUCH_OBJECT,NULL,0,NULL,0)
    | ;
33779|          /*lint -restore */
33780|          break;
33781|      }
33782|      case PSM_ERROR_CANT_CREATE_CACHE_FILE:
    | {
33783|          /*lint -save -e740 */
33784|
    | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
    | R_CANT_CREATE_CACHE_FILE,PSM_ERROR_CANT_CREATE_CACHE_FIL
    | E,NULL,0,NULL,0);
33785|          /*lint -restore */
33786|          break;
33787|      }
33788|      case STATUS_REQUEST_ABORTED :
33789|      case STATUS_PROCESS_IS_TERMINATING :
33790|      case PSM_CANCELED_BY_USER: {
33791|          /*lint -save -e740 */
33792|
    | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_CANC
    | ELED_BY_USER,PSM_CANCELED_BY_USER,NULL,0,NULL,0);
33793|          /*lint -restore */
33794|          break;
33795|      }
33796|      case PSM_EVALUATION_EXPIRED: {
33797|          /*lint -save -e740 */
33798|
    | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_EVAL
    | UATION_EXPIRED,PSM_EVALUATION_EXPIRED,NULL,0,NULL,0);

```

```

33799|          /*lint -restore */
33800|          break;
33801|      }
33802|      case PSM_INSUFFICIENT_CACHE: {
33803|          /*lint -save -e740 */
33804|
33805|          | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_INSU
33806|          | FFICIENT_CACHE,PSM_INSUFFICIENT_CACHE,NULL,0,NULL,0);
33807|          /*lint -restore */
33808|          break;
33809|      }
33810|      case PSM_ERROR_CACHEFILE_FULL: {
33811|          /*lint -save -e740 */
33812|
33813|          | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
33814|          | R_CACHEFILE_FULL,PSM_ERROR_CACHEFILE_FULL,NULL,0,NULL,0)
33815|          | ;
33816|          /*lint -restore */
33817|          break;
33818|      }
33819|      case
33820|          | PSM_MAXIMUM_ALLOWED_SNAPSHOTS_REACHED : {
33821|              WCHAR ErrorStr[12];
33822|              WCHAR *Strings[1];
33823|
33824|              | swprintf(ErrorStr,L"%d",PersistentDictionary::QueryMaxNu
33825|              | mUserSnapShots());
33826|              Strings[0] = ErrorStr;
33827|
33828|          /*lint -save -e740 */
33829|
33830|          | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_MAXI
33831|          | MUM_ALLOWED_SNAPSHOTS_REACHED,PSM_MAXIMUM_ALLOWED_SNAPSH
33832|          | OTS_REACHED,NULL,0,Strings,1);
33833|          /*lint -restore */
33834|          break;
33835|      }
33836|      case PSM_ERROR_VOLUME_FULL:
33837|      case STATUS_DISK_FULL : {
33838|          /*lint -save -e740 */
33839|
33840|          | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
33841|          | R_VOLUME_FULL,PSM_ERROR_VOLUME_FULL,NULL,0,NULL,0);
33842|          /*lint -restore */
33843|          break;
33844|      }
33845|
33846|      default: {
33847|          WCHAR ErrorStr[10];
33848|          WCHAR *Strings[1];

```

```

33836|
33837|         | swprintf(ErrorStr,L"%08x",Status);
33838|         Strings[0] = ErrorStr;
33839|         Debug(DEBUG_DCPSM,("SbOpenPSM:
33840|         | Logging error %08x\n",Status));
33841|         /*lint -save -e740 */
33842|         | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_SNAP
33843|         | SHOT_COULD_NOT_BE_CREATED,Status,NULL,0,Strings,1);
33844|         /*lint -restore */
33845|         break;
33846|     }
33847| }
33848|     | __except(ExceptionFilter(GetExceptionInformation())) {
33849|         Status = GetExceptionCode();
33850|         Debug(DEBUG_DCPSM,("Exception %08x in
33851|         | SbOpenPSM\n",Status));
33852|     }
33853|
33854|     UpdateGlobalStatus(PSM_IDLE);
33855|
33856|     if ( OpenCloseAcquired ) {
33857|         ReleaseOpenCloseResource();
33858|         OpenCloseAcquired = false;
33859|     }
33860|
33861|     return Status;
33862| }
33863|
33864| //-----
33865| | -----
33866| | --
33867|
33868| STATIC NTSTATUS InitVDiskThreads()
33869| {
33870|     NTSTATUS ntStatus;
33871|     HANDLE TempHandle=NULL;
33872|     KEVENT TempEvent={0};
33873|
33874|     | Reg_GetULONGKey(&gRegistryPath,L"VDiskIOHandling",0x0000
33875|     | ,&gVDiskIOHandling);
33876|
33877|     ntStatus = InitVDiskWriteCache( 0 );
33878|     if ( !NT_SUCCESS(ntStatus) ) {
33879|         Debug(DEBUG_DCPSM,("Error %08x initing vdisk
33880|         | write cache\n",ntStatus));

```

```

33875|     return ntStatus;
33876| }
33877|
33878| // mutex for thread functions
33879| ExInitializeFastMutex(&VDiskThreadMutex);
33880|
33881| //KeInitializeSemaphore ( &TdromWorkingSemaphore,
    | 1, 1 );
33882|
33883| KeInitializeEvent ( &VDiskExitingEvent,
33884|     NotificationEvent, //
    | type (notification or sync)
33885|     FALSE
    | // signaled
33886| );
33887|
33888|
33889| // init read thingys
33890| KeInitializeSpinLock(&ReadVDiskSpinLock );
33891| InitializeListHead( &ReadVDiskQueue );
33892| KeInitializeSemaphore ( &ReadVDiskSemaphore, 0,
    | MAXLONG );
33893|
33894| ntStatus = pmStartThread(
33895|     | (PKSTART_ROUTINE)ReadVDiskThread, // IN PKSTART_ROUTINE
    | StartRoutine,
33896|     (PVOID)0,
    | // IN PVOID StartContext
33897|     &TempHandle
    | // OUT PHANDLE ThreadHandle,
33898| );
33899|
33900| if ( NT_SUCCESS(ntStatus) ) {
33901|     // We dont use the handle, so get rid of it
33902|     ZwClose(TempHandle);
33903|     TempHandle = NULL;
33904|
33905|     // init write thingys
33906|     KeInitializeSpinLock(&WriteVDiskSpinLock );
33907|     InitializeListHead( &WriteVDiskQueue );
33908|     KeInitializeSemaphore ( &WriteVDiskSemaphore,
    | 0, MAXLONG );
33909|
33910|     KeInitializeEvent ( &TempEvent,
    | NotificationEvent, FALSE );
33911|
33912|     ntStatus = pmStartThread(
33913|         | (PKSTART_ROUTINE)WriteVDiskThread, // IN

```

```

    | PKSTART_ROUTINE StartRoutine,
33914|         (PVOID)&TempEvent,
    | // IN PVOID StartContext
33915|         &TempHandle
    | // OUT PHANDLE ThreadHandle,
33916|         );
33917|     if ( NT_SUCCESS(ntStatus) ) {
33918|         PVOID ObjectTable[3] = { &TempEvent,
    | &VdiskExitingEvent, &PSManExitingEvent};
33919|
33920|         // We dont use the handle, so get rid of it
33921|         ZwClose(TempHandle);
33922|         TempHandle=NULL;
33923|         ntStatus =
    | pmWaitForMultipleObjects(ObjectTable,3,NULL);
33924|
33925|         if ( ntStatus==STATUS_WAIT_0 ) {
33926|             Debug(DEBUG_DCPSM,("Vdisk thread
    | started\n"));
33927|
33928|             ntStatus = pmStartThread(
33929|
    | (PKSTART_ROUTINE)SendWriteAfterReadThread, // IN
    | PKSTART_ROUTINE StartRoutine,
33930|             (PVOID)0,
    | // IN PVOID StartContext
33931|             &TempHandle
    | // OUT PHANDLE ThreadHandle,
33932|             );
33933|
33934|             if ( NT_SUCCESS(ntStatus) ) {
33935|                 ZwClose(TempHandle);
33936|                 Debug(DEBUG_DCPSM,("All vdisk
    | threads started\n"));
33937|             } else {
33938|                 Debug(DEBUG_DCPSM,("Error %08x
    | starting WriteAfterRead Completion
    | thread\n",ntStatus));
33939|             }
33940|         } else {
33941|             Debug(DEBUG_DCPSM,("Exiting event
    | recieved\n"));
33942|             // exiting event is set
33943|             ntStatus = PSM_CANCELED_BY_USER;
33944|             // wait for event to be set before
    | continueing otherwise the thread will access
33945|             // the event and bsod
33946|             pmWaitForSingleObject(&TempEvent,NULL);
33947|         }
33948|     }

```

```

33949| }
33950|
33951| if ( !NT_SUCCESS(ntStatus) ) {
33952|     // tell the threads that did init to exit
33953|     pmSetEvent( &VDiskExitingEvent );
33954|     DelnitVDiskWriteCache();
33955| }
33956|
33957| return ntStatus;
33958| }
33959|
33960| //-----
    | -----
    | --
33961|
33962| NTSTATUS InitWorkerThreads()
33963| {
33964|     ULONG ThreadCount=0;
33965|     NTSTATUS ntStatus=STATUS_SUCCESS;
33966|
33967|     ThreadsAwake = 0;
33968|
33969|     ThreadObjects = (PMY_THREAD)
        | MemAllocatePoolWithTag(
33970|         | PagedPool,
33971|         | MaxThreads * sizeof(tMY_THREAD),
33972|         | THREADTAG
33973|         | );
33974|
33975|     if ( !ThreadObjects ) {
33976|         Debug(DEBUG_DCPM,("Error! No memory for thread
            | objects\n"));
33977|         return STATUS_INSUFFICIENT_RESOURCES;
33978|     }
33979|
33980|     // clear out array
33981|     RtlZeroMemory( ThreadObjects, MaxThreads *
        | sizeof(tMY_THREAD) );
33982|
33983|     ASSERT(GlobalThreadCount<=MaxThreads);
33984|     ASSERT(NumberOfThreads<=MaxThreads);
33985|     for (
        | ThreadCount=0;ThreadCount<NumberOfThreads;ThreadCount++
        | ) {
33986|         HANDLE TempHandle=NULL;
33987|

```



```

33988|     ntStatus = pmStartThread(
33989|
33990|         | (PKSTART_ROUTINE)SaveOriginalDataThread, // IN
33991|         | PKSTART_ROUTINE StartRoutine,
33992|         (PVOID)ThreadCount,
33993|         | // IN PVOID StartContext
33994|         &TempHandle
33995|         | // OUT PHANDLE ThreadHandle,
33996|         );
33997|
33998|     if ( !NT_SUCCESS(ntStatus) ) {
33999|         break;
34000|     }
34001|
34002|     ntStatus = ObReferenceObjectByHandle(
34003|         TempHandle,
34004|         | // IN HANDLE Handle,
34005|         | THREAD_ALL_ACCESS, // IN ACCESS_MASK DesiredAccess,
34006|         NULL,
34007|         | // IN POBJECT_TYPE ObjectType,
34008|         | /* optional */
34009|         | (KPROCESSOR_MODE)KernelMode, // IN
34010|         | KPROCESSOR_MODE AccessMode,
34011|         &ThreadObjects[ThreadCount].ThreadObject, // OUT PVOID
34012|         | *Object,
34013|         NULL
34014|         | // OUT POBJECT_HANDLE_INFORMATION HandleInformation
34015|         | /* optional */
34016|         );
34017|     //Debug(DEBUG_DCPSM,("Thread %d Handle = %08x,
34018|         | Object=%08x,
34019|         | Status=%08x\n",ThreadCount,TempHandle,ThreadObjects[Thre
34020|         | adCount].ThreadObject,ntStatus));
34021|     // Dont need the handle anymore now that we
34022|         | have an object
34023|     ZwClose(TempHandle);
34024|     TempHandle=NULL;
34025|     // exit if we dont have an object
34026|     if ( !NT_SUCCESS(ntStatus) ) {
34027|         break;
34028|     }
34029| }
34030|
34031| if ( !NT_SUCCESS(ntStatus) ) {
34032|     goto ThreadsFailed;
34033| } else {
34034|     HANDLE TempHandle=NULL;

```

```

34020|
34021|     WriteThreadObject.ThreadObject = NULL;
34022|     // start the thread that gets passed the irp..
34023|     ntStatus = pmStartThread(
34024|         | (PKSTART_ROUTINE)WriteDispatchThread, // IN
          | PKSTART_ROUTINE StartRoutine,
34025|         | (PVOID)0,
          | // IN PVOID StartContext
34026|         | &TempHandle
          | // OUT PHANDLE ThreadHandle,
34027|         | );
34028|
34029|     if ( NT_SUCCESS(ntStatus) ) {
34030|
34031|         ntStatus = ObReferenceObjectByHandle(
34032|             | TempHandle, // IN HANDLE Handle,
34033|             | THREAD_ALL_ACCESS, // IN ACCESS_MASK DesiredAccess,
34034|             | NULL,
          | // IN POBJECT_TYPE ObjectType,
          | /* optional */
34035|             | (KPROCESSOR_MODE)KernelMode, // IN
          | KPROCESSOR_MODE AccessMode,
34036|             | &WriteThreadObject.ThreadObject, // OUT PVOID *Object,
34037|             | NULL
          | // OUT POBJECT_HANDLE_INFORMATION HandleInformation
          | /* optional */
34038|             | );
34039|         Debug(DEBUG_DCPSM,("WriteDispatchThread
          | Handle = %08x, Object=%08x,
          | Status=%08x\n",TempHandle,WriteThreadObject.ThreadObject
          | ,ntStatus));
34040|         // Dont need the handle anymore now that we
          | have an object
34041|         ZwClose(TempHandle);
34042|         TempHandle=NULL;
34043|         // exit if we dont have an object
34044|     }
34045| }
34046|
34047| if ( !NT_SUCCESS(ntStatus) ) {
34048|     if ( WriteThreadObject.ThreadObject ) {
34049|
          | ObDereferenceObject(WriteThreadObject.ThreadObject);
34050|         WriteThreadObject.ThreadObject = NULL;
34051|     }

```

```

34052|     ThreadsFailed:
34053|     Debug(DEBUG_DCPsm,("Error! Unable to create
| threads\n"));
34054|
34055|     // close all objects to threads we created, so
| NT can destroy
34056|     // the objects, Make sure to grab resource in
| case threads are running
34057|
34058|     pmSetEvent( &VDiskExitingEvent );
34059|
34060|     pmAcquireMutex ( &WorkerThreadMutex, NULL );
34061|     ASSERT(GlobalThreadCount<=MaxThreads);
34062|     ASSERT(NumberOfThreads<=MaxThreads);
34063|     for (
| ThreadCount=0;ThreadCount<NumberOfThreads;ThreadCount++
| ) {
34064|         if (
| ThreadObjects[ThreadCount].ThreadObject ) {
34065|             ObDereferenceObject(ThreadObjects[ThreadCount].ThreadObj
| ect);
34066|             ThreadObjects[ThreadCount].ThreadObject
| = NULL;
34067|         }
34068|     }
34069|
34070|     FREE_POINTER(ThreadObjects);
34071|     pmReleaseMutex ( &WorkerThreadMutex );
34072| }
34073|
34074| return ntStatus;
34075| }
34076|
34077| //-----
| -----
| --
34078|
34079| NTSTATUS SbPreDelInit()
34080| {
34081|     PsmActive = 0;
34082|
34083|     // tell the threads that did init to exit
34084|     pmSetEvent( &VDiskExitingEvent );
34085|     pmSetEvent( &PSManExitingEvent );
34086|
34087|     // if any threads still running, wait for them to
| exit first
34088|     // most likely another request came in before the
| first

```

```

34089| // had a chance to finish
34090| if ( GlobalThreadCount ) {
34091|     LARGE_INTEGER TimeToWait={0};
34092|
34093|     Debug(DEBUG_DCPSM,("SbPreDeInit: Waiting for
| worker threads to exit before continueing\n"));
34094|     TimeToWait.QuadPart = RELATIVE(SECONDS(1));
34095|
34096|     // wait for threads to exit
34097|     while ( GlobalThreadCount ) {
34098|         Debug(DEBUG_DCPSM,("SbPreDeInit: Waiting;
| GlobalThreadCount=%08x\n",GlobalThreadCount));
34099|         KeDelayExecutionThread(
34100|             | (KPROCESSOR_MODE)KernelMode, // IN KPROCESSOR_MODE
| WaitMode,
34101|             FALSE, // IN
| BOOLEAN Alertable,
34102|             &TimeToWait // IN
| PLARGE_INTEGER Interval
34103|             );
34104|     }
34105|
34106|     Debug(DEBUG_DCPSM,("SbPreDeInit: Worker threads
| have all exited.\n"));
34107| }
34108|
34109|
34110| if ( PSManPSMInited ) {
34111|     DeInitVDiskWriteCache();
34112|
34113|     IrpDeInit();
34114|
34115|     PersistentDictionary::DeinitClass();
34116|
34117|     PSManPSMInited = 0;
34118| }
34119| return 0;
34120| }
34121|
34122| //-----
| -----
| --
34123|
34124| NTSTATUS
34125| SbPreInit(
34126|     pOT_USER User,
34127|     tOpenTransactionInInternal *Buffer,
34128|     PKEVENT AbortEvent
34129| )

```

```

34130| {
34131|     NTSTATUS          ntStatus=STATUS_SUCCESS;
34132|     ULONG              SectorCount=0;
34133|     ULONG              MaxSectorCount=0;
34134|
34135|     PAGED_CODE();
34136|     Debug(DEBUG_INFO,("PSM Preinit Called\n"));
34137|
34138|     SbGetRegistrySettings( &gRegistryPath );
34139|
34140| #ifndef NOPSM
34141|
34142|     if ( Buffer ) {
34143|         // Params are in MB, convert to number of
34144|         | sectors
34145|         SectorCount = Buffer->SizeOfCacheFileMB*2048;
34146|         | // FIXFIXFIX short cut this is actually
34147|         | // S = MB * 1024 *1024 / 512
34148|         MaxSectorCount =
34149|         | Buffer->MaxSizeOfCacheFileMB*2048;
34150|
34151|         /*
34152|         // OTM Fossil
34153|         if ( !(Buffer->InternalFlags &
34154|         | PSM_IFLAG_PERSISTENT) ) {
34155|             // make sure the cache file is at least 10
34156|             | megs big
34157|             // and that the max is greater than the
34158|             | current
34159|             if ( (SectorCount<(10*1024*1024)/512) ||
34160|             | (MaxSectorCount<SectorCount) ) {
34161|                 Debug(DEBUG_DCPSM,("Error! Invalid
34162|                 | cache file size\n"));
34163|                 return STATUS_INVALID_PARAMETER;
34164|             }
34165|         }
34166|         */
34167|     }
34168|
34169|     // if already open just say ok...
34170|     if ( PsmActive ) {
34171|         Debug(DEBUG_DCPSM,("PSM already open\n"));
34172|         goto PsmOpened;
34173|     }
34174|
34175|     // if already init'd, just say ok...
34176|     if ( PSMManPSMInit'd ) {
34177|         Debug(DEBUG_DCPSM,("PSM already init'd\n"));
34178|         goto PsmOpened;
34179|     }

```

```

34172|
34173| // if any threads still running, wait for them to
    | exit first
34174| // most likely another request came in before the
    | first
34175| // had a chance to finish
34176| if ( GlobalThreadCount ) {
34177|     LARGE_INTEGER TimeToWait={0};
34178|
34179|     Debug(DEBUG_DCPSM,("SbPreInit: Waiting for
    | worker threads to exit before continueing\n"));
34180|     TimeToWait.QuadPart = RELATIVE(SECONDS(1));
34181|
34182|     // wait for threads to exit
34183|     while ( GlobalThreadCount ) {
34184|         Debug(DEBUG_DCPSM,("SbPreInit: Waiting;
    | GlobalThreadCount=%08x\n",GlobalThreadCount));
34185|         KeDelayExecutionThread(
34186|             | (KPROCESSOR_MODE)KernelMode, // IN KPROCESSOR_MODE
    | WaitMode,
34187|             FALSE, // IN
    | BOOLEAN Alertable,
34188|             &TimeToWait // IN
    | PLARGE_INTEGER Interval
34189|             );
34190|     }
34191|
34192|     Debug(DEBUG_DCPSM,("SbPreInit: Worker threads
    | have all exited.\n"));
34193| }
34194|
34195|
34196| MaxWriteQueueDepth = 0;
34197| WriteQueueDepth = 0;
34198| LastErrorStatus = 0;
34199|
34200| if ( Buffer ) {
34201|     PSMAN_PSM_FLAGS = Buffer->Flags;
34202| }
34203|
34204|
34205| // InitAllTrees ( PSMAN_DRIVER_OBJECT );
34206|
34207| KeInitializeSemaphore ( &ThreadSemaphore, 0,
    | MAXLONG );
34208|
34209| KeInitializeEvent ( &WorkerThreadEvent,
    | NotificationEvent, FALSE );
34210|

```

```

34211|   KeInitializeEvent ( &Thread0Initd,
      | NotificationEvent, FALSE );
34212|
34213|   // mutex for Worker thread
34214|   ExInitializeFastMutex(&WorkerThreadMutex);
34215|
34216|   // Cache threshold mutex
34217|   ExInitializeFastMutex(&CacheThresholdMutex);
34218|
34219|   KeInitializeSemaphore ( &WriteSemaphore, 0, MAXLONG
      | );
34220|   KeInitializeSemaphore ( &WriteAfterReadSemaphore,
      | 0, MAXLONG );
34221|
34222|   KeInitializeEvent ( &PSManExitingEvent,
      | NotificationEvent, FALSE );
34223|
34224|   KeInitializeSpinLock(&ThreadsWorkToDoSpinLock );
34225|   InitializeListHead( &ThreadsWorkToDoQueue );
34226|
34227|   KeInitializeSpinLock(&WriteAfterReadSpinLock );
34228|   InitializeListHead( &WriteAfterReadQueue );
34229|
34230|   KeInitializeSpinLock(&WriteSpinLock );
34231|   InitializeListHead( &WriteQueue );
34232|   InitializeListHead( &ProcessingQueue );
34233|
34234|   IrpInit();
34235|
34236|   if ( Buffer ) {
34237|       ntStatus = VerifyGoodVolumeList(Buffer);
34238|   } else {
34239|       ntStatus = STATUS_SUCCESS;
34240|   }
34241|   if ( NT_SUCCESS(ntStatus) ) {
34242|       ntStatus = PersistentDictionary::InitClass (
34243|           | PSM_NORMAL_STAGE,
34244|           | Buffer,
34245|           | AbortEvent );
34246|
34247|       if ( NT_SUCCESS(ntStatus) ) {
34248|           // now that we got the cache file start up
34249|           | vdisk
34249|           ntStatus = InitWorkerThreads();
34250|
34251|           if ( NT_SUCCESS( ntStatus ) ) {
34252|

```

```

34253|         ntStatus = InitVdiskThreads();
34254|         if ( NT_SUCCESS(ntStatus) ) {
34255|             // PSM is completely initied
34256|             DumpDebugPointers ();
34257|             PsmOpened:
34258|             Debug(DEBUG_INFO,("PSM Preinit
| done\n"));
34259|             PSManPSMInited = TRUE;
34260|             return STATUS_SUCCESS;
34261|         } else {
34262|             Debug(DEBUG_DCPSM,("Error %08x
| initing vdisk threads\n",ntStatus));
34263|         }
34264|         pmSetEvent( &VdiskExitingEvent );
34265|
34266|         DeInitVdiskWriteCache();
34267|     } else {
34268|         Debug(DEBUG_DCPSM,("Error! unable to
| start worker threads\n"));
34269|         ntStatus =
| STATUS_INSUFFICIENT_RESOURCES;
34270|     }
34271|     // tell the threads that did init to exit
34272|     pmSetEvent( &PSManExitingEvent );
34273| } else {
34274|     Debug(DEBUG_DCPSM,("Error! InitClass failed
| %08x\n",ntStatus));
34275| }
34276| } else {
34277|     Debug(DEBUG_DCPSM,("Error! VerifyGoodVolumeList
| failed %08x\n",ntStatus));
34278| }
34279|
34280| pmSetEvent( &PSManExitingEvent );
34281| IrpDeInit();
34282| #endif
34283|
34284| return ntStatus;
34285| }
34286|
34287| //-----
| -----
| --
34288|
34289| void PsmOff()
34290| {
34291|     __try {
34292|         PDEVICE_OBJECT DevObj=NULL;
34293|         PFILTERED_EXTENSION DevExt=NULL;
34294|

```



```

34295|     Debug(DEBUG_PROCCALL,("PsmOff called\n"));
34296|     // no need to get global resource as any writes
| that occur before we turn it
34297|     // off will be handled when the threads are
| told to exit.
34298|     DevObj = PSMANDriverObject->DeviceObject;
34299|     while ( DevObj ) {
34300|         // if a filtered disk
34301|         if (
| PsmGetObjectype(DevObj)==OBJECT_FILTEREDDISK ) {
34302|             DevExt = GetFilteredExtension(DevObj);
34303|             if ( DevExt->PSMed ) {
34304|                 // we shouldnt be called with
| active unless something bad happened
34305|                 ASSERT(FALSE);
34306|                 DevExt->PSMed      = 0;
34307|                 DevExt->SignalRead  = 0;
34308|                 DevExt->SignalWrite = 0;
34309|                 DevExt->OpenCount   = 0;
34310|             }
34311|         } // if filtered disk
34312|
34313|         DevObj = DevObj->NextDevice;
34314|     } // while(DevObj)
34315| } __except
| (ExceptionFilter(GetExceptionInformation())) {
34316|     Debug(DEBUG_DCPSM,("PsmOff: Exception
| %08x\n",GetExceptionCode()));
34317| }
34318|
34319| PsmActive = FALSE;
34320| return;
34321| }
34322|
34323|
34324| //-----
| -----
| --
34325| //  clean up all sym links our dll didnt (because it
| was killed)
34326|
34327| void CleanupSymLinks( )
34328| {
34329|     WCHAR      DriveName[20] = L"\\DosDevices\\C:";
34330|     UNICODE_STRING UniName={0};
34331|     OBJECT_ATTRIBUTES ObjAttr={0};
34332|     HANDLE      SymHandle=NULL;
34333|     WCHAR      SymSpace[256]={0};
34334|     NTSTATUS     Status=STATUS_UNSUCCESSFUL;
34335|     ULONG        Ret=0;

```

```

34336|  UCHAR          DriveLetter=0;
34337|
34338|  PAGED_CODE();
34339|
34340|  Debug(DEBUG_PROCCALL|DEBUG_DCPSM,("CleanupSymlinks
    | Called\n"));
34341|
34342|  // get rid of virtual drive letters
34343|  for ( DriveLetter=2;DriveLetter<26;DriveLetter++ )
    | {
34344|
34345|      DriveName[12] = DriveLetter+65;
34346|
34347|      RtlInitUnicodeString( &UniName, DriveName );
34348|
34349|      ObjAttr.Length          =
    | sizeof(ObjAttr);
34350|      ObjAttr.RootDirectory   = NULL;
34351|      ObjAttr.Attributes      =
    | OBJ_CASE_INSENSITIVE;
34352|      ObjAttr.ObjectName      = &UniName;
34353|      ObjAttr.SecurityDescriptor = NULL;
34354|      ObjAttr.SecurityQualityOfService = NULL;
34355|
34356|      Status = ZwOpenSymbolicLinkObject( &SymHandle,
    | STANDARD_RIGHTS_READ, &ObjAttr );
34357|      if ( NT_SUCCESS(Status) ) {
34358|          WCHAR PsmDirName[35]={0};
34359|
34360|          | swprintf(PsmDirName,L"\\Device\\PsmDevices_%04x\\",PSM_L
    | OW_COMPATIBLE_VERSION);
34361|
34362|          UniName.Length = 0;
34363|          UniName.MaximumLength = 256;
34364|          UniName.Buffer = SymSpace;
34365|          Status =
    | ZwQuerySymbolicLinkObject(SymHandle,&UniName,&Ret);
34366|          if ( NT_SUCCESS(Status) ) {
34367|              Debug(DEBUG_DCPSM,("Symlink '%S' ==
    | '%wZ'\n",DriveName,&UniName));
34368|
34369|              if (
    | _wcsnicmp(UniName.Buffer,PsmDirName,wcslen(PsmDirName))==
    | =0 ) {
34370|                  Debug(DEBUG_DCPSM,("Cleaning up
    | Symlink '%wZ'\n",&UniName));
34371|                  // we found one!
34372|                  // Free it
34373|                  ZwClose(SymHandle);

```

```

34374|          SymHandle=NULL;
34375|          RtlInitUnicodeString (&UniName,
| DriveName);
34376|          IoDeleteSymbolicLink (&UniName);
34377|          continue;
34378|      }
34379|  } else {
34380|      Debug(DEBUG_DCPSM,("Error
| %08x\n",Status));
34381|  }
34382|
34383|      ZwClose(SymHandle);
34384|      SymHandle = NULL;
34385|  } else {
34386|      //Debug(DEBUG_DCPSM,("Error
| %08x\n",Status));
34387|  }
34388|  } // for
34389|
34390| // TODO FIXFIXFIX !FIX!FIX!FIX Remove volumes that do
| not have a drive letter.
34391|
34392| return;
34393| }
34394|
34395| //-----
| -----
| --
34396| // clean up all sym links our dll didnt (because it
| was killed)
34397|
34398| void CleanupSymLinksForDevice( WCHAR *NTDeviceName )
34399| {
34400|     WCHAR      DriveName[20] = L"\\DosDevices\\C:";
34401|     UNICODE_STRING UniName={0};
34402|     OBJECT_ATTRIBUTES ObjAttr={0};
34403|     HANDLE      SymHandle=NULL;
34404|     WCHAR      SymSpace[256]={0};
34405|     NTSTATUS      Status=STATUS_UNSUCCESSFUL;
34406|     ULONG      Ret=0;
34407|     WCHAR
| DriveMap[]=L"0123456789~`!#$%*-_+[]{}'ABDEFGHIJKLMNOPQR
| STUVWXYZ"; // C is skipped!!
34408|     WCHAR *DriveIndex=DriveMap;
34409|
34410|     PAGED_CODE();
34411|
34412|     Debug(DEBUG_PROCCALL|DEBUG_DCPSM,("CleanupSymlinks
| Called\n"));
34413|

```

```

34414| // get rid of virtual drive letters
34415| for ( ;*DriveIndex;DriveIndex++) {
34416|
34417|     DriveName[12] = *DriveIndex;
34418|
34419|     RtlInitUnicodeString( &UniName, DriveName );
34420|
34421|     ObjAttr.Length          =
        | sizeof(ObjAttr);
34422|     ObjAttr.RootDirectory   = NULL;
34423|     ObjAttr.Attributes      =
        | OBJ_CASE_INSENSITIVE;
34424|     ObjAttr.ObjectName       = &UniName;
34425|     ObjAttr.SecurityDescriptor = NULL;
34426|     ObjAttr.SecurityQualityOfService = NULL;
34427|
34428|     Status = ZwOpenSymbolicLinkObject( &SymHandle,
        | STANDARD_RIGHTS_READ, &ObjAttr );
34429|     if ( NT_SUCCESS(Status) ) {
34430|         UniName.Length = 0;
34431|         UniName.MaximumLength = 256;
34432|         UniName.Buffer = SymSpace;
34433|         Status =
            | ZwQuerySymbolicLinkObject(SymHandle,&UniName,&Ret);
34434|         if ( NT_SUCCESS(Status) ) {
34435|             ULONG DidOnce=0;
34436|
34437|             DoCheck:
34438|             Debug(DEBUG_DCPSM,("Symlink '%S' ==
                | '%wZ'\n",DriveName,&UniName));
34439|
34440|             if (
                | _wcsnicmp(UniName.Buffer,NTDeviceName,wcslen(NTDeviceNam
                | e))==0 ) {
34441|                 Debug(DEBUG_DCPSM,("Cleaning up
                    | Symlink '%wZ'\n",&UniName));
34442|                 // we found one!
34443|                 // Free it
34444|                 ZwClose(SymHandle);
34445|                 SymHandle=NULL;
34446|                 RtlInitUnicodeString (&UniName,
                    | DriveName);
34447|                 IoDeleteSymbolicLink (&UniName);
34448|                 continue;
34449|             } else {
34450|                 // see if the link is a sym link
34451|                 // ie, 1: == \??\Volume{00..00}
34452|                 if ( !DidOnce ) {
34453|                     DidOnce=1;
34454|                     if ( GetNTNameFromWin32Name(

```

```

    | UniName.Buffer, UniName.Buffer )==0 ) {
34455|         goto DoCheck;
34456|     }
34457| }
34458| }
34459| } else {
34460|     Debug(DEBUG_DCPSM,("Error
    | %08x\n",Status));
34461| }
34462|
34463|     ZwClose(SymHandle);
34464|     SymHandle = NULL;
34465| } else {
34466|     //Debug(DEBUG_DCPSM,("Error
    | %08x\n",Status));
34467| }
34468| } // for
34469|
34470| // TODO FIXFIXFIX !FIX!FIX!FIX Remove volumes that do
    | not have a drive letter.
34471|
34472| return;
34473| }
34474|
34475| //-----
    | -----
    | --
34476| // Free all snapshots in system
34477| //
34478| void FreeSnapShotResources()
34479| {
34480|     PDEVICE_OBJECT DevObj=NULL;
34481|     PFILTERED_EXTENSION DevExt=NULL;
34482|     pkSnapShotEntry p;
34483|
34484|     // start at the top
34485|     DevObj = PSMAN_DRIVER_OBJECT->DeviceObject;
34486|
34487|     while ( DevObj != NULL ) {
34488|         if (
            | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
34489|             DevExt = GetFilteredExtension(DevObj);
34490|
34491|             __try {
34492|                 GetSnapShotForWrite();
34493|             } __try {
34494|
            | p=GetTopSnapShot(&DevExt->SnapShots);
34495|                 while ( p ) {
34496|

```

```

    | FreeResourcesForVolume(DevObj,p);
34497|         DoneWithSnapShot(p);
34498|
    | p=GetTopSnapShot(&DevExt->SnapShots);
34499|         }
34500|         } __finally {
34501|             ReleaseSnapShotForWrite();
34502|         }
34503|     }
    | __except(ExceptionFilter(GetExceptionInformation())) {
34504|
    | Debug(DEBUG_DCPSM,("FreeSnapShotResources: Exception
    | %08x for device %08x\n",GetExceptionCode(),DevObj));
34505|     }
34506| }
34507|     DevObj = DevObj->NextDevice;
34508| }
34509| // FIXFIXFIX Free snapshots in user structures
    | !FIX!FIX
34510| }
34511|
34512| //-----
    | -----
    | --
34513|
34514| void MarkAllSnapShotsWithError( pkSnapShotEntry
    | SnapShot, NTSTATUS Error )
34515| {
34516|     PDEVICE_OBJECT   DevObj=NULL;
34517|     PFILTERED_EXTENSION DevExt=NULL;
34518|     pkSnapShotEntry p = NULL;
34519|
34520|     // start at the top
34521|     DevObj = PSMAN_DRIVER_OBJECT->DeviceObject;
34522|
34523|     __try {
34524|         while ( DevObj != NULL ) {
34525|             if (
    | PsmGetObjectTypeInfo(DevObj)==OBJECT_FILTEREDDISK ) {
34526|                 DevExt = GetFilteredExtension(DevObj);
34527|
34528|                 if ( SnapShot ) {
34529|                     // snapshot resource is should
    | already be acquired.
34530|
    | p=GetTopSnapShot(&DevExt->SnapShots);
34531|                     while ( p ) {
34532|                         p->MasterSnapShot->Status =
    | Error;
34533|

```

```

    | p=GetNextSnapShot(&DevExt->SnapShots,p);
34534|         }
34535|     } else {
34536|         // snapshot resource is not
    | acquired.
34537|         GetSnapShotForRead();
34538|         __try {
34539|
    | p=GetTopSnapShot(&DevExt->SnapShots);
34540|         while ( p ) {
34541|             p->MasterSnapShot->Status =
    | Error;
34542|
    | p=GetNextSnapShot(&DevExt->SnapShots,p);
34543|         }
34544|     } __finally {
34545|         ReleaseSnapShotForRead();
34546|     }
34547|
34548|     }
34549| }
34550|     DevObj = DevObj->NextDevice;
34551| }
34552| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
34553|     Debug(DEBUG_DCPSM,("MarkAll: Exception %08x for
    | device %08x\n",GetExceptionCode(),DevObj));
34554| }
34555| }
34556|
34557| //-----
    | -----
    | --
34558|
34559| void MarkAllSnapShotsWithErrorForVolume(
    | pkSnapShotEntry SnapShot, NTSTATUS Error )
34560| {
34561|     PDEVICE_OBJECT  DevObj=NULL;
34562|     PFILTERED_EXTENSION DevExt=NULL;
34563|     pkSnapShotEntry p;
34564|
34565|     ASSERT(SnapShot);
34566|
34567|     // start at the top
34568|     DevObj = SnapShot->DeviceObject;
34569|
34570|     __try {
34571|         if ( DevObj != NULL ) {
34572|             ASSERT (
    | PsmGetObjectype(DevObj)==OBJECT_FILTEREDDISK );

```

```

34573|
34574|         DevExt = GetFilteredExtension(DevObj);
34575|
34576|         // snapshot resource should already be
        | acquired.
34577|         p=GetTopSnapShot(&DevExt->SnapShots);
34578|         while ( p ) {
34579|             p->MasterSnapShot->Status = Error;
34580|
34581|         | p=GetNextSnapShot(&DevExt->SnapShots,p);
34582|         }
34583|     }
34584| }
        | __except(ExceptionFilter(GetExceptionInformation())) {
34585|     Debug(DEBUG_DCPSPM,("MarkAll: Exception %08x for
        | device %08x\n",GetExceptionCode(),DevObj));
34586| }
34587| }
34588|
34589| //-----
        | -----
        | --
34590|
34591| void LogClose( pkSnapShotMaster MasterSnapShot)
34592| {
34593|     ULONG LogSize;
34594|
34595|     __try {
34596|         if ( MasterSnapShot ) {
34597|             if ( MasterSnapShot->Instance <
        | MAX_NUMBER_OF_SNAPSHOTS ) {
34598|                 if ( gLogOpenClose ) {
34599|                     ULONG *DumpData;
34600|                     WCHAR *Strings[9];
34601|                     WCHAR VolList[100];
34602|                     WCHAR ID[10];
34603|                     WCHAR Month[4];
34604|                     WCHAR Day[4];
34605|                     WCHAR Year[6];
34606|                     WCHAR Hour[4];
34607|                     WCHAR Minute[4];
34608|                     WCHAR Second[4];
34609|                     TIME_FIELDS tm;
34610|                     LARGE_INTEGER Local;
34611|
34612|                     wcscpy(VolList,L "");
34613|
        | MakeVolumeListString(MasterSnapShot,VolList,sizeof(VolLi
        | st));

```



```

34614|
| sprintf(ID,L"%08x",MasterSnapShot);
34615|
34616|
| ExSystemTimeToLocalTime(&MasterSnapShot->SnapShotTime,&L
| ocal);
34617|         RtlTimeToTimeFields(&Local,&tm);
34618|         sprintf(Month,L"%02d",tm.Month);
34619|         sprintf(Day,L"%02d",tm.Day);
34620|         sprintf(Year,L"%4d",tm.Year);
34621|         sprintf(Hour,L"%02d",tm.Hour);
34622|         sprintf(Minute,L"%02d",tm.Minute);
34623|         sprintf(Second,L"%02d",tm.Second);
34624|
34625|         wcscat(VolList,L":\\");
34626|
34627| #define NumDumpltems 7
34628|         ULONG
| SizeOfDumpData=NumDumpltems*sizeof(ULONG);
34629|
34630|         LogSize =
| ((sizeof(IO_ERROR_LOG_MESSAGE)-sizeof(IO_ERROR_LOG_PACKE
| T)) +
34631|         ERROR_LOG_MAXIMUM_SIZE)
| -
34632|
| sizeof(IO_ERROR_LOG_MESSAGE) -
34633|         NumBytes(ID) -
34634|         NumBytes(Month) -
34635|         NumBytes(Day) -
34636|         NumBytes(Year) -
34637|         NumBytes(Hour) -
34638|         NumBytes(Minute) -
34639|         NumBytes(Second) -
34640|         SizeOfDumpData-
| (sizeof(WCHAR) * 8);
34641|
34642|         if ( NumBytes(VolList)>=LogSize ) {
34643|             // volume list is too big
34644|             wcscpy(VolList,L"...");
34645|         }
34646|
34647|         LogSize-=NumBytes(VolList);
34648|
34649|         if (
| _wcsicmp(MasterSnapShot->UserSnapShotName,L"")==0 ) {
34650|             // just leave blank
34651|         } else {
34652|             if (
| NumBytes(MasterSnapShot->UserSnapShotName)<LogSize ) {

```

```

34653|
    | wcscat(VolList,MasterSnapShot->UserSnapShotName);
34654|         } else {
34655|             WCHAR
    | *p=wcsrchr(MasterSnapShot->UserSnapShotName,L'\\');
34656|             if ( p ) {
34657|                 p++;
34658| #define DOT_SIZE (unsigned)(3*sizeof(WCHAR))
34659|
34660|             if (
    | (NumBytes(p)+DOT_SIZE)<LogSize ) {
34661|
    | wcscat(VolList,L"...");
34662|                 wcscat(VolList,p);
34663|             } else {
34664|                 Trunc:
34665|
    | wcscat(VolList,L"...");
34666|             if (
    | LogSize>DOT_SIZE ) {
34667|                 p =
    | MasterSnapShot->UserSnapShotName +
    | NumBytes(MasterSnapShot->UserSnapShotName) - LogSize -
    | 1;
34668|
    | p+=(DOT_SIZE/sizeof(WCHAR));
34669|
    | wcscat(VolList,p);
34670|             }
34671|         }
34672|     } else {
34673|         goto Trunc;
34674|     }
34675| }
34676| }
34677|
34678| #define NumStrings 8
34679|     Strings[0] = ID;
    | // 2
34680|     Strings[1] = VolList;
    | // 3
34681|     Strings[2] = Month;
    | // 4
34682|     Strings[3] = Day;
    | // 5
34683|     Strings[4] = Year;
    | // 6
34684|     Strings[5] = Hour;
    | // 7
34685|     Strings[6] = Minute;

```

```

| // 8
34686|         Strings[7] = Second;
| // 9
34687|
34688|         DumpData = (ULONG *)
| MemAllocatePoolWithTag(PagedPool,SizeOfDumpData,TEMPTAG)
| ;
34689|
34690|         if ( DumpData ) {
34691|             ULONG PerAt=0, PerHigh=0,
| PerUsed=0;
34692|
34693| //
| PersistentDictionary::GetVolumeSpaceUsed(PerAt,PerHigh,P
| erUsed);
34694|
34695|         DumpData[0] =
| (ULONG)MasterSnapShot;
34696|         DumpData[1] = PerUsed;
34697|         DumpData[2] = PerAt;
34698|         DumpData[3] = PerHigh;
34699|
34700|         DumpData[4] =
| MaxWriteQueueDepth;
34701|         DumpData[5] = LastErrorStatus;
34702|         DumpData[6] = NumberOfThreads;
34703|     } else {
34704|         SizeOfDumpData = 0;
34705|     }
34706| #undef NumDumpltems
34707|
34708|         /*lint -save -e740 */
34709|
| LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_CLOS
| ED_INFORMATION,0,DumpData,SizeOfDumpData,Strings,NumStri
| ngs);
34710|         /*lint -restore */
34711|
34712|         if ( DumpData ) {
34713|             FREE_POINTER(DumpData);
34714|         }
34715|     }
34716| }
34717| }
34718| }
| __except(ExceptionFilter(GetExceptionInformation())) {
34719|     Debug(DEBUG_DCPSM,("LogClose: Exception
| %08x\n",GetExceptionCode()));
34720| }
34721| }

```

```

34722|
34723| typedef struct sCloseCallBack {
34724|     pOT_USER User;
34725|     pkSnapShotMaster MasterSnapShot;
34726|     NTSTATUS Status;
34727| } tCloseCallBack, *pCloseCallBack;
34728|
34729| void CloseCallBack( pCloseCallBack CallBack )
34730| {
34731|     CallBack->Status =
        | InternalClosePSM(CallBack->User, CallBack->MasterSnapShot
        | );
34732|     PsTerminateSystemThread( 0 );
34733| }
34734|
34735| //-----
        | -----
        | --
34736| // must be called with AcquireOpenCloseResource();
34737| NTSTATUS InternalClosePSM( pOT_USER User,
        | tkSnapShotMaster *MasterSnapShot )
34738| {
34739|     ULONG      ThreadCount=0;
34740|     NTSTATUS    Status = STATUS_SUCCESS;
34741|     NTSTATUS    CountStatus = STATUS_SUCCESS;
34742|     ULONG      NumActiveSnapShots = 0;
34743|
34744|     PAGED_CODE();
34745|     if ( GlobalSystemProcessId != PsGetCurrentProcess()
        | ) {
34746|         // in a different process id, lets spawn off a
        | thread
34747|         // and call us back in the system process.
        | This is
34748|         // so we can access our virtual memory
34749|         HANDLE TempHandle;
34750|         tCloseCallBack CallBack;
34751|
34752|         CallBack.User=User;
34753|         CallBack.MasterSnapShot = MasterSnapShot;
34754|         pmStartThread(
34755|             (PKSTART_ROUTINE)CloseCallBack,
        | // IN PKSTART_ROUTINE StartRoutine,
34756|             (PVOID)&CallBack,
        | // IN PVOID StartContext
34757|             &TempHandle
        | // OUT PHANDLE ThreadHandle,
34758|             );
34759|         ZwWaitForSingleObject(TempHandle,FALSE,NULL);
34760|         ZwClose(TempHandle);

```

```

34761|     return CallBack.Status;
34762| }
34763|
34764|     ASSERT(User != NULL);
34765|
34766|     __try {
34767|         Debug(DEBUG_DCPSM,("InternalClosePSM:
    | User=%08x,
    | MasterSnapShot=%08x\n",User,MasterSnapShot));
34768|
34769| #ifdef DEBUG
34770|     if (
    | pmExamineSemaphore(&PSMOpenCloseSemaphore)>0 ) {
34771|         // event not acquired!
34772|         Debug(DEBUG_DCPSM,("InternalClosePSM:
    | OpenClose resource not acquired!\n"));
34773|         DbgBreakPoint();
34774|     }
34775| #endif
34776|
34777|     if(!MasterSnapShot) {
34778|         if ( User ) {
34779|             if ( User->Open ) {
34780|                 // remove last snapshot since they
    | didnt specify
34781|                 PLIST_ENTRY ListEntry =
    | User->SnapShots.Flink;
34782|                 if ( ListEntry!&User->SnapShots )
    | {
34783|                     /*lint -save -e413 */
34784|                     pkSnapshotEntry p =
    | CONTAINING_RECORD( ListEntry, tkSnapshotEntry, User);
34785|                     /*lint -restore */
34786|
    | MasterSnapShot=p->MasterSnapShot;
34787|
    | Debug(DEBUG_INFO,("InternalClosePSM: MasterSnapShot is
    | null, using %08x\n",MasterSnapShot));
34788|                 } else {
34789|                     ASSERT(FALSE);
34790|                 }
34791|             } else {
34792|
    | Debug(DEBUG_INFO,("InternalClosePSM: MasterSnapShot is
    | null, but User->Open is zero.\n"));
34793|             }
34794|         } else {
34795|             Debug(DEBUG_INFO,("InternalClosePSM:
    | MasterSnapShot is null, but User is null too.\n"));
34796|         }

```

```

34797|     }
34798|
34799|     //ASSERT(MasterSnapShot);
34800|
34801|     if ( GlobalData->NumActive>0 ) {
34802|         InterlockedDecrement((PLONG)
34803|         | &GlobalData->NumActive);
34804|         Debug(DEBUG_DCPSM,("InternalClosePSM:
34805|         | Decremented NumActive to
34806|         | %08x\n",GlobalData->NumActive));
34807|     } else {
34808|         Debug(DEBUG_DCPSM,("InternalClosePSM:
34809|         | NumActive == 0!\n"));
34810|     }
34811|
34812|     if ( User ) {
34813|         if ( User->Open ) {
34814|             InterlockedDecrement((PLONG)
34815|             | &User->Open);
34816|         } else {
34817|             Debug(DEBUG_DCPSM,("InternalClosePSM:
34818|             | User doesnt have us open!\n"));
34819|             // can happen if open failed
34820|             | (sbopenpsm)
34821|         }
34822|
34823|         if ( User->ErrorEvent ) {
34824|             // dont need this any more.
34825|             ObDereferenceObject(User->ErrorEvent);
34826|             User->ErrorEvent = NULL;
34827|         }
34828|
34829|         if ( User->Open ) {
34830|             // still has active opens.
34831|             Debug(DEBUG_INFO,("InternalClosePSM:
34832|             | User still has active opens %08x\n",User->Open));
34833|             // get rid of this snapshot
34834|             RemoveUserLinks(MasterSnapShot);
34835|             LogClose(MasterSnapShot);
34836|             GetSnapShotForWrite();
34837|             __try {
34838|                 | GetRidOfSpecificSnapShotForUser(User,MasterSnapShot);
34839|             } __finally {
34840|                 | ASSERT(IsListEmpty(&MasterSnapShot->SnapShots));
34841|                 ReleaseSnapShotForWrite();
34842|             }
34843|             VDiskUnMapInDrives(MasterSnapShot);
34844|             FREE_POINTER(MasterSnapShot);

```

```

34837|          // ASSERT(User->NumOpenSnapShots>0);
34838|          try_return(Status = STATUS_SUCCESS);
34839|      }
34840|
34841|          // delete the drives
34842|          if ( MasterSnapShot ) {
34843|              RemoveUserLinks(MasterSnapShot);
34844|              LogClose(MasterSnapShot);
34845|              Debug(DEBUG_DCPSM,("InternalClosePSM:
| Freeing all volumes for user %08x SnapShot
| %08x!\n",User,MasterSnapShot));
34846|
| FreeVolumesForUser(User,MasterSnapShot);
34847|
34848|              VDiskUnMapInDrives(MasterSnapShot);
34849|              // incase they didnt call
| CloseExclusive
34850|
| DeInitForExclusive(User,MasterSnapShot);
34851|              FREE_POINTER(MasterSnapShot);
34852|          }
34853|
34854|          ASSERT(User->NumOpenSnapShots==0);
34855|          User->NumOpenSnapShots = 0;
34856|      } else {
34857|          // no user specified, just try and clean up
34858|          Debug(DEBUG_DCPSM,("InternalClosePSM: No
| user specified\n"));
34859|      }
34860|
34861|          // only close if all people are gone.
34862|          if ( GlobalData->NumActive>0 ) {
34863|              Debug(DEBUG_INFO,("InternalClosePSM: Still
| active psm users\n"));
34864|              try_return(Status = STATUS_SUCCESS);
34865|          }
34866|
34867|          // At this point NO more users are using Psm.
34868|          // lets deinit and free resources
34869|
34870| #ifdef DEBUG
34871|          DumpAllDisks();
34872| #endif
34873|
34874| #if 1
34875|          // TESTTEST
34876|          Debug(DEBUG_DCPSM,("InternalClosePSM: No users
| left, removing volumes!\n"));
34877|          // disable drives before turning off psm.
34878|          VDiskUnMapInAllDrives();

```

```

34879|
34880|     PsmOff();
34881|
34882|     if ( !PSManPSMInited ) {
34883|         Debug(DEBUG_DCPSM,("InternalClosePsm: Psm
| not initialized, nothing to cleanup!\n"));
34884|         try_return(Status=STATUS_SUCCESS);
34885|     }
34886|
34887|     PSManPSMInited = FALSE;
34888|
34889|     Debug(DEBUG_INFO,("PSM Statistics:\n"));
34890|     {
34891|         ULONG PerAt=0, PerHigh=0, PerUsed=0;
34892|
34893|         //
| PersistentDictionary::GetVolumeSpaceUsed(PerAt,PerHigh,P
| erUsed);
34894|
34895|         Debug(DEBUG_INFO,("  Cache File Size =
| %d/%d (%d)\n",PerUsed,PerAt,PerHigh));
34896|     }
34897|     Debug(DEBUG_INFO,("  Max Depth Queue =
| %d\n",MaxWriteQueueDepth));
34898|     Debug(DEBUG_INFO,("  Last Error    =
| %08x\n",LastErrorStatus));
34899|
34900|     // free any snapshots that may still be hanging
| around
34901|     FreeSnapShotResources();
34902|
34903|     // wait for system to quiesce
34904|     while ( OutstandingRequests ) {
34905|         LARGE_INTEGER TimeToWait={0};
34906|
34907|         Debug(DEBUG_DCPSM,("InternalClosePSM:
| Waiting for ssystem to quiesce\n"));
34908|         TimeToWait.QuadPart = RELATIVE(SECONDS(1));
34909|
34910|         while ( OutstandingRequests ) {
34911|             KeDelayExecutionThread(
| (KPROCESSOR_MODE)KernelMode, FALSE, &TimeToWait );
34912|         }
34913|     }
34914|
34915|     // have tdrom threads exit.
34916|     pmSetEvent( &VDiskExitingEvent );
34917|
34918|     // tell worker threads to exit
34919|     pmSetEvent( &PSManExitingEvent );

```



```

34920|
34921| #ifndef NOPSM
34922|
34923|     // if any threads to cleanup after...
34924|     if ( GlobalThreadCount ) {
34925|         LARGE_INTEGER TimeToWait={0};
34926|         Debug(DEBUG_DCPSM,("InternalClosePSM:
    | Waiting for worker threads to exit\n"));
34927|         TimeToWait.QuadPart = RELATIVE(SECONDS(1));
34928|
34929|         // wait for threads to exit
34930|         while ( GlobalThreadCount ) {
34931|             KeDelayExecutionThread(
34932|
    | (KPROCESSOR_MODE)KernelMode, // IN KPROCESSOR_MODE
    | WaitMode,
34933|
    | FALSE,
    | // IN BOOLEAN Alertable,
34934|
    | &TimeToWait //
    | IN PLARGE_INTEGER Interval
34935|
    | );
34936|     }
34937| }
34938|
34939| if ( VDiskNumberOfThreads ) {
34940|     LARGE_INTEGER TimeToWait={0};
34941|
34942|     Debug(DEBUG_DCPSM,("InternalClosePSM:
    | Waiting for vdisk threads to exit\n"));
34943|     TimeToWait.QuadPart = RELATIVE(SECONDS(1));
34944|
34945|     // wait for threads to exit
34946|     while ( VDiskNumberOfThreads ) {
34947|         KeDelayExecutionThread(
34948|
    | (KPROCESSOR_MODE)KernelMode, // IN KPROCESSOR_MODE
    | WaitMode,
34949|
    | FALSE,
    | // IN BOOLEAN Alertable,
34950|
    | &TimeToWait //
    | IN PLARGE_INTEGER Interval
34951|
    | );
34952|     }
34953| }
34954|
34955|     // free any reads to the volume that may have
    | occurred
34956|     while ( !IsListEmpty(&ReadVDiskQueue) ) {
34957|         PLIST_ENTRY ListEntry=NULL;
34958|         tReadRequest *ReadRequest=NULL;

```

```

34959|         PIRP Irp=NULL;
34960|         KIRQL oldIrql;
34961|
34962|         Debug(DEBUG_DCPSM,("Cleaning up read on
| vdisk queue\n"));
34963|         ListEntry = ExInterlockedRemoveHeadList (
34964|             | &ReadVDiskQueue,          // List Head
34965|             | &ReadVDiskSpinLock        // Lock
34966|             );
34967|         /*lint -save -e413 */
34968|         ReadRequest = CONTAINING_RECORD( ListEntry,
| tReadRequest, ListEntry );
34969|         /*lint -restore */
34970|
34971|         Irp      = ReadRequest->Irp;
34972|         pmAcquireSpinLock(&WriteSpinLock,&oldIrql);
34973|
| RemoveEntryList(&(ReadRequest->ProcessingEntry));
34974|         pmReleaseSpinLock(&WriteSpinLock,oldIrql);
34975|         FREE_POINTER(ReadRequest);
34976|
34977|         Irp->IoStatus.Status =
| STATUS_NO_MEDIA_IN_DEVICE;
34978|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
34979|     }
34980|
34981|     // free any writes to the volume
34982|     while ( !IsListEmpty(&WriteVDiskQueue) ) {
34983|         PLIST_ENTRY ListEntry;
34984|         tWriteRequest *WriteRequest;
34985|         KIRQL oldIrql;
34986|         PIRP Irp;
34987|
34988|         Debug(DEBUG_DCPSM,("Cleaning up write on
| vdisk queue\n"));
34989|         ListEntry = ExInterlockedRemoveHeadList (
34990|             | &WriteVDiskQueue,          // List Head
34991|             | &WriteVDiskSpinLock        // Lock
34992|             );
34993|         /*lint -save -e413 */
34994|         WriteRequest = CONTAINING_RECORD(
| ListEntry, tWriteRequest, ListEntry );
34995|         /*lint -restore */
34996|
34997|         Irp      = WriteRequest->Irp;
34998|         pmAcquireSpinLock(&WriteSpinLock,&oldIrql);

```

```

34999|
| RemoveEntryList(&(WriteRequest->ProcessingEntry));
35000|         pmReleaseSpinLock(&WriteSpinLock,oldIrql);
35001|         FREE_POINTER(WriteRequest);
35002|
35003|         Irp->IoStatus.Status =
| STATUS_NO_MEDIA_IN_DEVICE;
35004|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
35005|     }
35006|
35007|     DelnitVDiskWriteCache();
35008|
35009|     // if anything on the queue, empty it, as no
| threads are running to do it
35010|     while ( !IsListEmpty(&ThreadsWorkToDoQueue) ) {
35011|         PIRP NIrp = SbGetWorkItem();
35012|
35013|         if ( NIrp ) {
35014|             Debug(DEBUG_DCPSM,("InternalClosePSM:
| Cleaning up write on threads queue\n"));
35015|
35016|             __try {
35017|                 PIO_STACK_LOCATION CurrentStackLoc
| = IoGetCurrentIrpStackLocation( NIrp );
35018|
35019|
35020|                 // free buffer that we allocated
35021|                 /*lint -save -e613 */
35022|
| FREE_POINTER((PVOID)(CurrentStackLoc->Parameters.Others.
| Argument1));
35023|                 /*lint -restore */
35024|             }
| __except(ExceptionFilter(GetExceptionInformation())) {
35025|                 Status = GetExceptionCode();
35026|                 Debug(DEBUG_INFO,("Exception %08x
| trying to free pointer\n",Status));
35027|             }
35028|
35029|             //free irp we allocated
35030|             IrpFreeIrp(NIrp);
35031|         }
35032|     }
35033|
35034|     // complete writes that may have been sent to
| our writer thread
35035|     if ( !IsListEmpty(&WriteQueue) ) {
35036|         Debug(DEBUG_DCPSM,("InternalClosePSM:
| Cleaning up writes on write queue\n"));
35037|         SbCompleteWritesOnQueue();

```

```

35038|     }
35039|
35040|     if ( WriteThreadObject.ThreadObject ) {
35041|
35042|         | ObDereferenceObject(WriteThreadObject.ThreadObject);
35043|         WriteThreadObject.ThreadObject = NULL;
35044|     }
35045|     if ( ThreadObjects ) {
35046|         // close all objects to threads we created,
35047|         | so NT can destory
35048|         // the objects
35049|         ASSERT(GlobalThreadCount<=MaxThreads);
35050|         ASSERT(NumberOfThreads<=MaxThreads);
35051|         for (
35052|             | ThreadCount=0;ThreadCount<NumberOfThreads;ThreadCount++
35053|             | ) {
35054|                 if (
35055|                     | ThreadObjects[ThreadCount].ThreadObject ) {
35056|                         | ObDereferenceObject(ThreadObjects[ThreadCount].ThreadObj
35057|                         | ect);
35058|
35059|                         | ThreadObjects[ThreadCount].ThreadObject = NULL;
35060|                     }
35061|                 }
35062|             FREE_POINTER( ThreadObjects );
35063|         }
35064|         PersistentDictionary::DeinitClass();
35065|         PSMAN_PSM_FLAGS = 0;
35066|         #if TRACK_CONTENTIONS
35067|             pmDumpStatistics();
35068|         #endif
35069|         IrpDeInit();
35070|         CleanupSymLinks();
35071|         #endif
35072|         #if 1
35073|             Debug(DEBUG_DCPSM,("InternalClosePSM: Zeroing
35074|             | out structures used during PSM\n"));
35075|             // Clean out stuff so if we access them again,
35076|             | we will fault.
35077|             //
35078|             | RtlZeroMemory(&VDiskExitingEvent,sizeof(VDiskExitingEven
35079|             | t));
35080|

```

```

    | RtlZeroMemory(&ReadVDiskSpinLock,sizeof(ReadVDiskSpinLoc
    | k));
35076|
    | RtlZeroMemory(&ReadVDiskQueue,sizeof(ReadVDiskQueue));
35077|
    | RtlZeroMemory(&ReadVDiskSemaphore,sizeof(ReadVDiskSemaph
    | ore));
35078|
    | RtlZeroMemory(&WriteVDiskSpinLock,sizeof(WriteVDiskSpinL
    | ock));
35079|
    | RtlZeroMemory(&WriteVDiskQueue,sizeof(WriteVDiskQueue));
35080|
    | RtlZeroMemory(&WriteVDiskSemaphore,sizeof(WriteVDiskSema
    | phore));
35081|
    | RtlZeroMemory(&ThreadSemaphore,sizeof(ThreadSemaphore));
35082|
    | RtlZeroMemory(&WorkerThreadEvent,sizeof(WorkerThreadEven
    | t));
35083|
    | RtlZeroMemory(&Thread0Inited,sizeof(Thread0Inited));
35084|
    | RtlZeroMemory(&WorkerThreadMutex,sizeof(WorkerThreadMute
    | x));
35085|
    | RtlZeroMemory(&CacheThresholdMutex,sizeof(CacheThreshold
    | Mutex));
35086|
    | RtlZeroMemory(&WriteSemaphore,sizeof(WriteSemaphore));
35087|
    | RtlZeroMemory(&WriteAfterReadSemaphore,sizeof(WriteAfter
    | ReadSemaphore));
35088|    //
    | RtlZeroMemory(&PSManExitingEvent,sizeof(PSManExitingEven
    | t));
35089|
    | RtlZeroMemory(&ThreadsWorkToDoSpinLock,sizeof(ThreadsWor
    | kToDoSpinLock));
35090|
    | RtlZeroMemory(&ThreadsWorkToDoQueue,sizeof(ThreadsWorkTo
    | DoQueue));
35091|
    | RtlZeroMemory(&WriteAfterReadSpinLock,sizeof(WriteAfterR
    | eadSpinLock));
35092|
    | RtlZeroMemory(&WriteAfterReadQueue,sizeof(WriteAfterRead
    | Queue));
35093|
    | RtlZeroMemory(&WriteSpinLock,sizeof(WriteSpinLock));

```

```

35094|     RtlZeroMemory(&WriteQueue,sizeof(WriteQueue));
35095|
35096|     RtlZeroMemory(&ProcessingQueue,sizeof(ProcessingQueue));
35097| #endif
35098| #endif
35099|     try_exit: NOTHING;
35100| }
35101| | __except(ExceptionFilter(GetExceptionInformation())) {
35102|     Status = GetExceptionCode();
35103|     Debug(DEBUG_INFO,("Exception %08x in
35104| | InternalClosePSM\n",Status));
35105| }
35106| MemTrackPrintStats( "ClosePSM" );
35107| MemShowUsage();
35108| return Status;
35109| }
35110| //-----
35111| | -----
35112| | --
35113| NTSTATUS ValidateKernelSnapShotPointer ( PVOID
35114| | KernelPointer )
35115| {
35116|     NTSTATUS status = STATUS_SUCCESS;
35117|     Debug(DEBUG_DCPSPM,("Entering
35118| | ValidateKernelSnapShotPointer(%p)\n",KernelPointer));
35119|     tPSM_GetPersistentSnapShotsOut *list =
35120|     (tPSM_GetPersistentSnapShotsOut
35121|     | *)MemAllocatePoolWithTag (
35122|     | PagedPool,
35123|     | sizeof(tPSM_GetPersistentSnapShotsOut),
35124|     | TEMPTAG );
35125|     if ( list ) {
35126|         status = SbGetPersistentSnapShots (list);
35127|         if ( NT_SUCCESS(status) ) {
35128|             BOOLEAN foundit = FALSE;
35129|             for ( int i=0; i<MAX_NUMBER_OF_SNAPSHOTS &&
35130| | list->KernelPointers[i]; ++i ) {
35131|                 if ( KernelPointer ==
35132| | list->KernelPointers[i] ) {
35133|                     foundit = TRUE;

```

```

35131|
    | Debug(DEBUG_DCPSM,("ValidateKernelSnapShotPointer:
    | found at list[%d]\n",i));
35132|         break;
35133|     }
35134| }
35135|
35136|     if ( !foundit ) {
35137|         status = STATUS_INVALID_HANDLE;
35138|
    | Debug(DEBUG_DCPSM,("ValidateKernelSnapShotPointer: did
    | not find %p in list\n",KernelPointer));
35139|     } else {
35140|         // check to make sure it is a valid
    | handle
35141|         __try {
35142|             if (
    | MmlsAddressValid(KernelPointer) ) {
35143|                 ULONG Test =
    | ((pkSnapShotMaster)KernelPointer)->Instance;
35144|             } else {
35145|                 status = STATUS_INVALID_HANDLE;
35146|             }
35147|         }
    | __except(ExceptionFilter(GetExceptionInformation())) {
35148|
    | Debug(DEBUG_DCPSM,("ValidateKernelSnapShotPointer:
    | Exception %08x for Snapshot pointer %08x is not
    | valid\n",GetExceptionCode(),KernelPointer));
35149|         status = STATUS_INVALID_HANDLE;
35150|     }
35151| }
35152| }
35153|     FREE_POINTER(list);
35154| } else {
35155|     Debug(DEBUG_INFO,("Cannot allocate snapshot
    | list in ValidateKernelSnapShotPointer\n"));
35156|     status = STATUS_INSUFFICIENT_RESOURCES;
35157| }
35158|
35159| return status;
35160| }
35161|
35162| //-----
    | -----
    | --
35163|
35164| NTSTATUS
35165| SbClosePSM( tClosePSMInternal *Buffer )
35166| {

```

```

35167|  pOT_USER    User=NULL;
35168|  NTSTATUS Status=0;
35169|
35170|  PAGED_CODE();
35171|
35172|  Debug(DEBUG_INFO,("PSM Close Called %08x
    | %08x\n",PsGetCurrentProcess() , PsGetCurrentThread()));
35173|
35174|  Status = ValidateKernelSnapShotPointer (
    | Buffer->KernelSnapShotPointer );
35175|
35176|  if ( Status ) {
35177|      return Status;
35178|  }
35179|
35180|  User =
    | FindPSMUserBySnapShot((pkSnapShotMaster)Buffer->KernelSn
    | apShotPointer);
35181|  if ( !User ) {
35182|      /*lint -save -e740 */
35183|      User = FindPSMUser( PsGetCurrentProcess(),
    | PsGetCurrentThread() );
35184|      /*lint -restore */
35185|      if ( !User ) {
35186|          Debug(DEBUG_DCPSM,("PSM can not find
    | user!!!! failing close %08x,
    | %08x\n",PsGetCurrentProcess(), PsGetCurrentThread()));
35187|          return STATUS_INVALID_HANDLE;
35188|      }
35189|  }
35190|
35191|  if ( !User->Open ) {
35192|      Debug(DEBUG_DCPSM,("SbClosePsm: User does not
    | have psm open!"));
35193|      return STATUS_INVALID_HANDLE;
35194|  }
35195|
35196|  UpdateGlobalStatus(PSM_DESTROYING_SNAPSHOT);
35197|
35198|  if (
    | AcquireOpenCloseResourceOnly(NULL)==STATUS_WAIT_0 ) {
35199|      __try {
35200|          if ( !Buffer->KernelSnapShotPointer ) {
35201|              // must be old client! so dec num users
35202|              ASSERT(GlobalData->NumActive);
35203|              InterlockedDecrement((PLONG)
    | &GlobalData->NumActive);
35204|              InterlockedDecrement((PLONG)
    | &User->Open);
35205|

```



```

35206|         Debug(DEBUG_DCPSM,("SbClosePSM:
| Decremented NumActive to %08x
| open=%08x\n",GlobalData->NumActive,User->Open));
35207|     }
35208|
| InternalClosePSM(User,(pkSnapShotMaster)Buffer->KernelSn
| apShotPointer);
35209|     } __finally {
35210|         ReleaseOpenCloseResource();
35211|     }
35212| } else {
35213|     Status = PSM_CANCELED_BY_USER;
35214| }
35215|
35216| UpdateGlobalStatus(PSM_IDLE);
35217|
35218| Debug(DEBUG_INFO,("SbClosePSM returning
| %08x\n",Status));
35219| //Debug (DEBUG_PROCCALL,("PSManDeviceControlObject
| Done\n")); ;
35220| return Status;
35221| }
35222|
35223| //-----
| -----
| --
35224|
35225| NTSTATUS SbInternalRevertBuffer (
35226|         PDEVICE_OBJECT
| DeviceObject,
35227|         PVDISK_EXTENSION
| VDiskExt,
35228|         ULARGE_INTEGER
| Sector,
35229|         ULONG
| Count,
35230|         ULONG
| /*Delete*/,
35231|         char
| *Buffer,
35232|         ULONG
| *SectorsChanged )
35233| {
35234|     ULONG CountDid=0;
35235|     ULARGE_INTEGER DS;
35236|     DS.QuadPart = (unsigned __int64)Count *
| VDiskExt->BPS;
35237| //     Debug(DEBUG_DICT,("Reverting %08x:%08x for
| %08x\n",VDiskExt->SnapShot->Dictionary,Sector,Count));
35238|     ASSERT(SectorsChanged != NULL);

```

```

35239|  *SectorsChanged = 0;
35240|
35241|  NTSTATUS Status =
    | VDiskExt->SnapShot->Dictionary->searchMultiple (
35242|      GetFilteredExtension(VDiskExt->PSMDevice),
35243|      Sector,
35244|      Count,
35245|      CountDid,
35246|      NULL,
35247|      DS,
35248|      Buffer,
35249|      gVDiskDoVirtualIO ? DICT_FLAG_VIRTUAL_IO : 0);
35250|
35251|  if ( Status==STATUS_NOT_FOUND ) {
35252|      Status = STATUS_SUCCESS;
35253|      *SectorsChanged = 0;
35254|  } else
35255|      if ( Status==STATUS_SUCCESS ) {
35256|          *SectorsChanged = CountDid;
35257|      }
35258|  return Status;
35259| }
35260|
35261| //-----
    | -----
    | --
35262|
35263| NTSTATUS FreeResourcesForVolume (
35264|     PDEVICE_OBJECT
    | DeviceBeingPSMed,
35265|     pkSnapShotEntry
    | SnapShot )
35266| {
35267|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DeviceBeingPSMed);
35268|     pkSnapShotMaster Master = NULL;
35269|
35270|     Debug(DEBUG_PROCCALL,("FreeResourcesForVolume
    | called\n"));
35271| #ifdef DEBUG
35272|     if ( !IsSnapShotAcquiredForWrite() ) {
35273|         Debug(DEBUG_DCPSM,("FreeResourcesForVolume:
    | Snapshot resource not acquired!\n"));
35274|         DbgBreakPoint();
35275|     }
35276| #endif
35277|
35278|     // get snapshot to use
35279|     if ( !SnapShot ) {
35280|         SnapShot = GetTopSnapShot(&DevExt->SnapShots);

```

```

35281|     if ( !SnapShot ) {
35282|         Debug(DEBUG_DCPSM,("Volume not
| snapshot\n"));
35283|         return STATUS_NO_SUCH_DEVICE;
35284|     }
35285|     Debug(DEBUG_DCPSM,("FreeResourcesForVolume: No
| snapshap passed in, using %08x\n",SnapShot));
35286|     DoneWithSnapShot(SnapShot);
35287| }
35288| #if 1
35289| // TESTTEST
35290|
35291| /*
35292|     Question: Should we move most of this to
| DoneWithSnapShot since most of this
35293|         is related to cleaning
| up after a snapshot when it is finished being used.
35294|         We could do the same
| thing since it is "reference" counted.
35295| */
35296|
35297| Master = SnapShot->MasterSnapShot;
35298| Debug(DEBUG_DCPSM,("FreeResourcesForVolume:
| SnapShot=%08x, master=%08x, Count=%08x,
| mc=%08x\n",SnapShot,Master,SnapShot->Count,Master->Count
| ));
35299|
35300| SnapShot->Deleted = TRUE;
35301|
35302| RemoveEntryList(&SnapShot->DevExt);
35303| InitializeListHead(&SnapShot->DevExt);
35304| RemoveEntryList(&SnapShot->Master);
35305| InitializeListHead(&SnapShot->Master);
35306|
35307| InterlockedDecrement((PLONG) &Master->Count);
35308|
35309| // this happens when mapping in hasnt run yet.
35310| if(SnapShot->Count==0) {
35311|     // bump up the reference count and decrement it
| again
35312|     // so the lower level code notices the count
| went to 0
35313|     // and cleans up this snapshot
35314|     Debug(DEBUG_DCPSM,("FreeResourcesForVolume:
| Deleting snapshots as mapping in hasnt occurred
| yet.\n"));
35315|     UseSnapShot(SnapShot);
35316|     DoneWithSnapShot(SnapShot);
35317| }
35318|

```

```

35319|   if ( DevExt->PSMed ) {
35320|       InterlockedDecrement((PLONG) &DevExt->PSMed);
35321|   } else {
35322|       Debug(DEBUG_DCPSM,("FreeResourcesForVolume:
| Error volume not referenced (psmed)!\n"));
35323|   }
35324|   if ( DevExt->OpenCount ) {
35325|       InterlockedDecrement((PLONG)
| &DevExt->OpenCount);
35326|   } else {
35327|       Debug(DEBUG_DCPSM,("FreeResourcesForVolume:
| Error volume not referenced!\n"));
35328|   }
35329|
35330| #ifdef DEBUG
35331|   if ( IsListEmpty(&DevExt->SnapShots) ) {
35332|       Debug(DEBUG_DCPSM,("FreeResourcesForVolume:
| Volume %08x has no more
| snapshots!\n", DeviceBeingPSMed));
35333|   }
35334|
35335| #endif
35336| #endif
35337|   return STATUS_SUCCESS;
35338| }
35339|
35340|
35341| //-----
| -----
| --
35342|
35343| NTSTATUS SbFreeVolume( pPSM_FreeVolume Volume )
35344| {
35345|   NTSTATUS Status=STATUS_SUCCESS;
35346|   PDEVICE_OBJECT DevObj=NULL;
35347|   PVDISK_EXTENSION DevExt=NULL;
35348|   pOT_USER User=NULL;
35349|   pkSnapShotEntry SavedSnapShot=NULL;
35350|
35351|   Debug(DEBUG_PROCCALL,("SbFreeVolume called\n"));
35352|   if ( AcquireOpenCloseResource()==STATUS_WAIT_0 ) {
35353|       __try {
35354|           /*lint -save -e740 */
35355|           User = FindPSMUser( PsGetCurrentProcess(),
| PsGetCurrentThread() );
35356|           /*lint -restore */
35357|           if ( !User ) {
35358|               Debug(DEBUG_DCPSM,("SbFreeVolume: PSM
| can not find user!!!! failing close %08x,
| %08x\n",PsGetCurrentProcess(), PsGetCurrentThread()));

```

```

35359|         return STATUS_INVALID_HANDLE;
35360|     }
35361|
35362|     if ( !User->Open ) {
35363|         Debug(DEBUG_DCPSM,("SbFreeVolume: User
| does not have psm open!"));
35364|         return STATUS_INVALID_HANDLE;
35365|     }
35366|
35367|     DevObj = GetObjectFromVDiskName(
| Volume->VolumeName );
35368|     if ( DevObj ) {
35369|         DevExt = GetVDiskExtension(DevObj);
35370|
35371|         // look for right snapshot on this
| device
35372|         __try {
35373|             GetSnapShotForWrite();
35374|             SavedSnapShot=NULL;
35375|             __try {
35376|                 // we save this pointer as when
| FreePSMVolume is done DevExt->SnapShot will be NULL
35377|                 SavedSnapShot =
| DevExt->SnapShot;
35378|                 UseSnapShot(SavedSnapShot);
35379|
| Debug(DEBUG_DCPSM,("SbFreeVolume: Freeing %08x %08x,
| '%S'\n",User,
| Volume->KernelSnapShotPointer,Volume->VolumeName));
35380|
| FreePSMVolume(User,DevExt->PSMDevice,SavedSnapShot);
35381|             } __finally {
35382|                 if ( SavedSnapShot ) {
35383|
| DoneWithSnapShot(SavedSnapShot);
35384|                 }
35385|                 ReleaseSnapShotForWrite();
35386|             }
35387|         }
| __except(ExceptionFilter(GetExceptionInformation())) {
35388|             Debug(DEBUG_DCPSM,("SbFreeVolume:
| Exception %08x for device
| %08x\n",GetExceptionCode(),DevObj));
35389|         }
35390|     } else {
35391|         Debug(DEBUG_DCPSM,("SbFreeVolume:
| Unable to find object for '%S'\n",Volume->VolumeName));
35392|     }
35393| } __finally {
35394|     ReleaseOpenCloseResource();

```

```

35395|     }
35396| } else {
35397|     Status = PSM_CANCELED_BY_USER;
35398| }
35399|
35400| return Status;
35401| }
35402|
35403| //-----
| -----
| --
35404|
35405| NTSTATUS SbFreeRanges( pPSM_FreeRanges Ranges )
35406| {
35407|     PDEVICE_OBJECT DevObj=NULL;
35408|     PVDISK_EXTENSION DevExt=NULL;
35409|     ULONG Count;
35410|     ULONG i,j;
35411|     ULARGE_INTEGER UL;
35412|     pkSnapshotMaster
| Master=(pkSnapshotMaster)(Ranges->KernelSnapshotPointer)
| ;
35413|     NTSTATUS Err = ValidateKernelSnapshotPointer
| (Master);
35414|
35415|     if ( NT_SUCCESS(Err) ) {
35416|         // Debug(DEBUG_PROCCALL,("SbFreeRanges
| called %08x\n",Ranges));
35417|
35418|         if ( (Master->ExclusiveProcess!=0) &&
35419|             | (Master->ExclusiveProcess!=PsGetCurrentProcess()) ) {
35420|             Debug(DEBUG_DCPSM,("SbOpenPsm: Someone else
| (%08x) already has exclusive
| access!",Master->ExclusiveProcess));
35421|             return PSM_ERROR_LOCKED_EXCLUSIVE;
35422|         }
35423|         if ( Master->ExclusiveProcess==0 ) {
35424|             Debug(DEBUG_DCPSM,("Need exclusive!"));
35425|             return PSM_ERROR_EX_LOCK_NEEDED;
35426|         }
35427|
35428|         DevObj =
| GetObjectFromVdiskName(Ranges->VolumeName);
35429|         if ( DevObj ) {
35430|             DevExt = GetVdiskExtension(DevObj);
35431|
35432|             __try {
35433|                 for ( i=0;i<Ranges->NumberOfRanges;i++
| ) {

```

```

35434|
35435|
35436|
35437|         Count = Ranges->RangeList[i].Count;
35438|
35439|         GetSnapShotForRead();
35440|         __try {
35441|             if ( DevExt->SnapShot ) {
35442|
35443|                 | UseSnapShot(DevExt->SnapShot);
35444|
35445|                 __try {
35446|                     for ( j=0;j<Count;j++ )
35447|                         | {
35448|                             UL.QuadPart =
35449|                             | Ranges->RangeList[i].Offset.QuadPart+j;
35450|                             // FIXFIXFIX is the
35451|                             | above in bytes or sectors or granules????????
35452|
35453|                             // this routine
35454|                             | needs it in sectors
35455|
35456|                             | DevExt->SnapShot->Dictionary->searchAndDeleteSingle (
35457|
35458|                             | GetFilteredExtension(DevExt->PSMDevice),
35459|                             | UL );
35460|
35461|                             // FIXFIXFIX need
35462|                             | to add physical scanning here !FIX!FIX!FIX
35463|
35464|                             // free sector so
35465|                             | it isnt psmed again, regardless of it being in our
35466|                             | cache file
35467|
35468|                             if (
35469|                             | DevExt->SnapShot->PSMSectors ) {
35470|
35471|                             | PsmBitPositionValidate (DevExt->SnapShot->PSMSectors,
35472|                             | UL.LowPart);
35473|
35474|                             | RtlClearBits(DevExt->SnapShot->PSMSectors,UL.LowPart,1);
35475|
35476|                             }
35477|                             } // for j
35478|                             } __finally {
35479|
35480|                             | DoneWithSnapShot(DevExt->SnapShot);
35481|
35482|                             }
35483|                             Err = 0;
35484|                             } else {

```

```

35468|
| Debug(DEBUG_DCPSM,("FreeRanges: SnapShot has been
| deleted while waiting or not psmmed\n"));
35469|         Err =
| STATUS_NO_MEDIA_IN_DEVICE;
35470|     }
35471|     } __finally {
35472|         ReleaseSnapShotForRead();
35473|     }
35474|     } // for i
35475| }
| __except(ExceptionFilter(GetExceptionInformation())) {
35476|     Err = GetExceptionCode();
35477|     Debug(DEBUG_DCPSM,("FreeRanges:
| Exception %08x for device %08x\n",Err,DevObj));
35478| }
35479| } else {
35480|     Err = STATUS_INVALID_PARAMETER;
35481| }
35482| }
35483| #ifdef DEBUG
35484|     if ( Err ) {
35485|         Debug(DEBUG_DCPSM,("FreeRanges: Error %08x
| trying to free range\n",Err));
35486|     }
35487| #endif
35488|     return Err;
35489| }
35490|
35491| //-----
| -----
| --
35492|
35493| NTSTATUS SbGetProgress( pkSnapShotMaster
| MasterSnapShot, tPSM_GetProgressOut *Progress )
35494| {
35495|     NTSTATUS status = STATUS_SUCCESS;
35496|     ULONG PerAt=0, PerHigh=0, PerUsed=0;
35497|     LARGE_INTEGER CT = {0};
35498|
35499|     Progress->OutOfSeconds = SecondsToWait;
35500|
35501|     if ( MasterSnapShot ) {
35502|         status = ValidateKernelSnapShotPointer
| (MasterSnapShot);
35503|         if ( NT_SUCCESS(status) ) {
35504|             Progress->OutOfSeconds =
| MasterSnapShot->OutOfSeconds;
35505|         }
35506|     }

```



```

35507|
35508|     if ( NT_SUCCESS(status) ) {
35509|         KeQuerySystemTime(&CT);
35510|
35511|         if ( CurrentTime.QuadPart!=0 ) {
35512|             Progress->OnSecond =
35513|             | (ULONG)(CT.QuadPart-CurrentTime.QuadPart) / 1000 / 1000
35514|             | / 10;
35515|         } else {
35516|             Progress->OnSecond = 0;
35517|         }
35518| //
35519|     | PersistentDictionary::GetVolumeSpaceUsed(PerAt,PerHigh,P
35520|     | erUsed);
35521|
35522|     Progress->CurrentCacheFileSize = PerUsed;
35523|     Progress->CacheFileSize      = PerAt;
35524| }
35525|
35526| return status;
35527| }
35528| //-----
35529| | -----
35530| | --
35531|
35532| NTSTATUS SbGetPersistentSnapShots (
35533|     | tPSM_GetPersistentSnapShotsOut *Out )
35534| {
35535|     PDEVICE_OBJECT  DevObj=NULL;
35536|     PFILTERED_EXTENSION DevExt=NULL;
35537|     pkSnapshotEntry p;
35538|     NTSTATUS Status=STATUS_SUCCESS;
35539|     ULONG NumFound=0;
35540|
35541|     // start at the top
35542|     DevObj = PSMAN_DRIVER_OBJECT->DeviceObject;
35543|
35544|     __try {
35545|         RtlZeroMemory(Out,sizeof(*Out));
35546|
35547|         while ( DevObj != NULL ) {
35548|             if (
35549|                 | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
35550|                 DevExt = GetFilteredExtension(DevObj);
35551|
35552|                 GetSnapshotForRead();
35553|             }
35554|             __try {

```

```

35549|
| p=GetTopSnapShot(&DevExt->SnapShots);
35550|         while ( p ) {
35551|             // only store for each unique
| snapshot
35552|                 ULONG FoundMatch=FALSE;
35553|                 ULONG i;
35554|                 for ( i=0;i<NumFound;i++ ) {
35555|                     if (
| p->MasterSnapShot==Out->KernelPointers[i] ) {
35556|                         FoundMatch = TRUE;
35557|                         break;
35558|                     }
35559|                 }
35560|                 if ( !FoundMatch ) {
35561|
| Out->KernelPointers[NumFound++] = p->MasterSnapShot;
35562|                 }
35563|
35564|
| p=GetNextSnapShot(&DevExt->SnapShots,p);
35565|         }
35566|         } __finally {
35567|             ReleaseSnapShotForRead();
35568|         }
35569|     }
35570|     DevObj = DevObj->NextDevice;
35571| }
35572| }
| __except(ExceptionFilter(GetExceptionInformation())) {
35573|     Status = GetExceptionCode();
35574|     Debug(DEBUG_DCPsm,("SbGetPersistentSnapShots:
| Exception %08x\n",Status));
35575| }
35576|
35577| return Status;
35578| }
35579|
35580| //-----
| -----
| --
35581|
35582| NTSTATUS SbGetKernelSnapShotInfo( pkSnapShotMaster
| Master, tPSM_GetKernelSnapShotInfoOut *Out )
35583| {
35584|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
35585|     __try {
35586|         Status = ValidateKernelSnapShotPointer
| (Master);
35587|         if ( NT_SUCCESS(Status) ) {

```

```

35588|         Out->Instance = Master->Instance;
35589|         Out->SnapShotTime = Master->SnapShotTime;
35590|         Out->Status = Master->Status;
35591|         Out->DllPrivateUse = Master->DllPrivateUse;
35592|         Out->CallerPrivateUse =
            | (PVOID)(Master->GroupNumber);
35593|         Out->NumToKeep = Master->NumToKeep;
35594|         Out->Priority = Master->Priority;
35595|         Out->SnapShotFlags = Master->SnapShotFlags;
35596|         Out->Persistent = Master->Persistent;
35597|
            | wcsncpy(Out->UserSnapShotName,Master->UserSnapShotName);
35598|     }
35599| }
            | __except(ExceptionFilter(GetExceptionInformation())) {
35600|     Status = GetExceptionCode();
35601|     Debug(DEBUG_DCPSM,("SbGetKernelSnapShotInfo:
            | Exception %08x\n",Status));
35602| }
35603| return Status;
35604| }
35605|
35606| //-----
            | -----
            | --
35607|
35608| void RemoveCallBack( pCloseCallBack CallBack )
35609| {
35610|     __try {
35611|         CallBack->Status =
            | SbRemoveVirtualWrites(CallBack->MasterSnapShot);
35612|     }
            | __except(ExceptionFilter(GetExceptionInformation())) {
35613|         Debug(DEBUG_DCPSM,("RemoveCallBack: Exception
            | %08x\n",GetExceptionCode()));
35614|     }
35615|     PsTerminateSystemThread( 0 );
35616| }
35617|
35618| NTSTATUS SbRemoveVirtualWrites( pkSnapShotMaster Master
            | )
35619| {
35620|     if ( GlobalSystemProcessId != PsGetCurrentProcess()
            | ) {
35621|         // in a different process id, lets spawn off a
            | thread
35622|         // and call us back in the system process.
            | This is
35623|         // so we can access our virtual memory and do
            | disk io

```

```

35624|     HANDLE TempHandle;
35625|     tCloseCallBack CallBack;
35626|
35627|     CallBack.User=NULL;
35628|     CallBack.MasterSnapShot = Master;
35629|     pmStartThread(
35630|         (PKSTART_ROUTINE)RemoveCallBack,
35631|         | // IN PKSTART_ROUTINE StartRoutine,
35632|         (PVOID)&CallBack,
35633|         | // IN PVOID StartContext
35634|         &TempHandle
35635|         | // OUT PHANDLE ThreadHandle,
35636|         );
35637|     ZwWaitForSingleObject(TempHandle,FALSE,NULL);
35638|     ZwClose(TempHandle);
35639|     return CallBack.Status;
35640| }
35641|
35642| NTSTATUS Status = STATUS_SUCCESS;
35643| WCHAR Buffer [256];
35644|
35645| GetSnapShotForRead();
35646| EnableWritesToNewFiles();
35647| __try {
35648|     // loop though and remove virtual writes for
35649|     | all volumes
35650|     // this snapshot is of
35651|     pkSnapShotEntry s =
35652|     | GetTopSnapShotForMaster(&Master->SnapShots);
35653|     if ( s ) {
35654|         while ( s ) {
35655|             Status =
35656|             | ((pPersistentDictionary)s->Dictionary)->RemoveVirtualWri
35657|             | tes();
35658|             if ( Status == STATUS_SUCCESS ) {
35659|                 // Delete nested snapshots
35660|                 | directories from virtual disk...
35661|                 PFILTERED_EXTENSION DevExt =
35662|                 | GetFilteredExtension(s->DeviceObject);
35663|                 PDEVICE_OBJECT Virtual =
35664|                 | GetVdiskObjectForName(DevExt->Name,Master->Instance);
35665|                 // if virtual device exists
35666|                 | (because part2 has run), cleanup nested snapshots.
35667|                 if(Virtual) {
35668|                     PVDISK_EXTENSION VDisk =
35669|                     | GetVDiskExtension(Virtual);
35670|
35671|                     | swprintf(Buffer,L"\\Device\\PsmDevices_\\%04x\\%s_\\%d\\%s",
35672|

```

```

    | PSM_LOW_COMPATIBLE_VERSION,
35661|                VDisk->Name,
35662|                VDisk->Instance,
35663|                gSnapShotDirName );
35664|
35665|                SbSnapShotCleanup (Buffer);
35666|        }
35667|    } else {
35668|        DoneWithSnapShot(s);
35669|        break;
35670|    }
35671|
    | s=GetNextSnapShotForMaster(&Master->SnapShots,s);
35672|    }
35673|    } else {
35674|        Status = STATUS_NOT_FOUND;
35675|    }
35676|    } __finally {
35677|        DisableWritesToNewFiles();
35678|        ReleaseSnapShotForRead();
35679|    }
35680|    return Status;
35681| }
35682|
35683| //-----
    | -----
    | --
35684|
35685| void MakeVirutalWritesPersistentCallBack(
    | pCloseCallBack CallBack )
35686| {
35687|     CallBack->Status =
    | SbMakeVirtualWritesPersistent(CallBack->MasterSnapShot);
35688|     PsTerminateSystemThread( 0 );
35689| }
35690|
35691| //-----
    | -----
    | --
35692|
35693| NTSTATUS SbMakeVirtualWritesPersistent (
    | pkSnapShotMaster Master )
35694| {
35695|     if ( GlobalSystemProcessId != PsGetCurrentProcess()
    | ) {
35696|         // in a different process id, lets spawn off a
    | thread
35697|         // and call us back in the system process.
    | This is
35698|         // so we can access our virtual memory and do

```

```

    | disk io
35699|     HANDLE TempHandle;
35700|     tCloseCallBack CallBack;
35701|
35702|     CallBack.User=NULL;
35703|     CallBack.MasterSnapShot = Master;
35704|     pmStartThread(
35705|
    | (PKSTART_ROUTINE)MakeVirutalWritesPersistentCallBack,
35706|         (PVOID)&CallBack,
35707|         &TempHandle );
35708|
35709|     ZwWaitForSingleObject(TempHandle,FALSE,NULL);
35710|     ZwClose(TempHandle);
35711|     return CallBack.Status;
35712| }
35713|
35714| NTSTATUS Status = STATUS_SUCCESS;
35715| GetSnapShotForRead();
35716| __try {
35717|     // loop though and remove virtual writes for
    | all volumes
35718|     // this snapshot is of
35719|     pkSnapShotEntry s =
    | GetTopSnapShotForMaster(&Master->SnapShots);
35720|     if ( s ) {
35721|         while ( s ) {
35722|             Status =
    | ((pPersistentDictionary)s->Dictionary)->MakeVirtualWrite
    | sPersistent();
35723|             if ( !NT_SUCCESS(Status) ) {
35724|                 DoneWithSnapShot(s);
35725|                 break;
35726|             }
35727|
    | s=GetNextSnapShotForMaster(&Master->SnapShots,s);
35728|         }
35729|     } else {
35730|         Status = STATUS_NOT_FOUND;
35731|     }
35732| } __finally {
35733|     ReleaseSnapShotForRead();
35734| }
35735| return Status;
35736| }
35737|
35738| //-----
    | -----
    | --
35739|

```

```

35740| ULONG SnapShotFlagsAreValid ( CHAR SnapShotFlags )
35741| {
35742|     ULONG Valid = TRUE;
35743|
35744|     if ( PSM_SS_IsPersistent(SnapShotFlags) ) {
35745|         if ( SnapShotFlags &
35746|             | PSM_SS_BIT_SAVE_TEMP_ON_EXIT ) {
35747|             Valid = FALSE; // Save-Temp-On-Exit
35748|             | applies to temporary snapshots only
35749|         }
35750|         if ( PSM_SS_IsReadOnly(SnapShotFlags) ) {
35751|             if ( SnapShotFlags &
35752|                 | PSM_SS_BIT_VIRTUAL_WRITES_PERSISTENT ) {
35753|                 Valid = FALSE; // Persistent virtual
35754|                 | writes are valid only for persistent read-write
35755|                 | snapshots.
35756|             }
35757|         }
35758|     } else {
35759|         if ( SnapShotFlags &
35760|             | PSM_SS_BIT_VIRTUAL_WRITES_PERSISTENT ) {
35761|                 Valid = FALSE; // Persistent virtual
35762|                 | writes are valid only for persistent read-write
35763|                 | snapshots.
35764|             }
35765|     }
35766|     return Valid;
35767| }
35768|
35769| //-----
35770| | -----
35771| | --
35772|
35773| NTSTATUS SbSetKernelSnapShotInfo( pkSnapShotMaster
35774|     | Master, tPSM_SetKernelSnapShotInfo *Info )
35775| {
35776|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
35777|     __try {
35778|         Status = ValidateKernelSnapShotPointer
35779|             | (Master);
35780|         if ( NT_SUCCESS(Status) ) {
35781|             Master->GroupNumber =
35782|                 | (ULONG)(Info->CallerPrivateUse);
35783|             Master->NumToKeep = Info->NumToKeep;
35784|             Master->Priority = Info->Priority;
35785|
35786|             CHAR OldSnapShotFlags =
35787|                 | Master->SnapShotFlags;

```

```

35776|         CHAR NewSnapShotFlags =
| Info->SnapShotFlags;
35777|
35778|         // Verify that the flags are defined
| validly.
35779|         ASSERT (
| SnapShotFlagsAreValid(OldSnapShotFlags) );
35780|         if (
| !SnapShotFlagsAreValid(NewSnapShotFlags) ) {
35781|             Status = STATUS_INVALID_PARAMETER;
35782|
| Debug(DEBUG_DCPSM,("SbSetKernelSnapShotInfo: Invalid
| snapshot flags = %02x\n",NewSnapShotFlags));
35783|         } else {
35784|             // Also, don't allow persistent to
| change to temporary, or vice versa.
35785|             // We might allow this in the future,
| but it would open a few cans of worms...
35786|             // For one thing, we use two different
| C++ classes to represent each.
35787|             if (
| PSM_SS_IsPersistent(OldSnapShotFlags) ) {
35788|                 if (
| PSM_SS_IsTemporary(NewSnapShotFlags) ) {
35789|                     Status =
| STATUS_INVALID_PARAMETER;
35790|
| Debug(DEBUG_DCPSM,("SbSetKernelSnapShotInfo: Attempt
| to change persistent snapshot to temporary!\n"));
35791|                 }
35792|             } else if (
| PSM_SS_IsTemporary(OldSnapShotFlags) ) {
35793|                 Status = STATUS_INVALID_PARAMETER;
35794|
| Debug(DEBUG_DCPSM,("SbSetKernelSnapShotInfo: Attempt
| to modify temporary snapshot flags!\n"));
35795|             }
35796|
35797|             if ( NT_SUCCESS(Status) ) {
35798|                 Master->SnapShotFlags =
| NewSnapShotFlags;
35799|
35800|                 // update persistent dictionary
| database
35801|
| PersistentDictionary::BeginUpdate();
35802|                 __try {
35803|                     GetSnapShotForRead();
35804|                 } __try {
35805|                     pkSnapShotEntry s =

```



```

    | GetTopSnapShotForMaster(&Master->SnapShots);
35806|         if ( s ) {
35807|             while ( s ) {
35808|                 Status =
    | ((pPersistentDictionary)s->Dictionary)->SetSnapShotInfo(
    | Master);
35809|                 if ( Status ==
    | STATUS_SUCCESS ) {
35810|                     s =
    | GetNextSnapShotForMaster(&Master->SnapShots,s);
35811|                 } else {
35812|
    | DoneWithSnapShot(s);
35813|                     break;
35814|                 }
35815|             }
35816|         } else {
35817|             Status =
    | STATUS_NOT_FOUND;
35818|
    | Debug(DEBUG_DCPSM,("SbSetKernelSnapShotInfo: No
    | snapshot entries found in master=%08x\n",Master));
35819|         }
35820|     } __finally {
35821|         ReleaseSnapShotForRead();
35822|     }
35823| } __finally {
35824|
    | PersistentDictionary::EndUpdate();
35825|     }
35826|
35827|     if ( NT_SUCCESS(Status) ) {
35828|
    | Debug(DEBUG_DCPSM,("SbSetKernelSnapShotInfo:
    | OldFlags=%02x,
    | NewFlags=%02x\n",OldSnapShotFlags,NewSnapShotFlags));
35829|
    | // NOTE: We cannot get here
    | unless this is a persistent snapshot.
35831|         // No need to check again.
35832|         if (
    | PSM_SS_IsReadWrite(OldSnapShotFlags) &&
    | PSM_SS_IsReadOnly(NewSnapShotFlags) ) {
35833|
    | Debug(DEBUG_DCPSM,("Snapshot changing from writeable to
    | read-only; undoing virtual writes\n"));
35834|             Status =
    | SbRemoveVirtualWrites (Master);
35835|         } else if (
    | PSM_SS_IsReadOnly(OldSnapShotFlags) &&

```

```

    | PSM_SS_IsReadWrite(NewSnapShotFlags) ) {
35836|
    | Debug(DEBUG_DCPSM,("Snapshot changing to read-write
    | persistent; making existing virtual writes
    | persistent\n"));
35837|         Status =
    | SbMakeVirtualWritesPersistent (Master);
35838|         }
35839|     }
35840| }
35841| }
35842| }
35843| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
35844|     Status = GetExceptionCode();
35845|     Debug(DEBUG_DCPSM,("SbSetKernelSnapShotInfo:
    | Exception %08x\n",Status));
35846| }
35847| return Status;
35848| }
35849|
35850| //-----
    | -----
    | --
35851|
35852| NTSTATUS SbGetKernelSnapShotVolumes (
35853|     pkSnapShotMaster
    | Master,
35854|     WCHAR *Buffer,
35855|     ULONG *Return )
35856| {
35857|     NTSTATUS Status = STATUS_SUCCESS;
35858|     pkSnapShotEntry p = 0;
35859|     ULONG BufferSize = *Return;
35860|     ULONG Count=0;
35861|     *Return = 0;
35862|
35863|     __try {
35864|         Status = ValidateKernelSnapShotPointer
    | (Master);
35865|         if ( NT_SUCCESS(Status) ) {
35866|             GetSnapShotForRead();
35867|             __try {
35868|                 p =
    | GetTopSnapShotForMaster(&Master->SnapShots);
35869|                 ULONG BufferBytesLeft = BufferSize;
35870|                 while ( p ) {
35871|                     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(p->DeviceObject);
35872|                     ULONG LenChars =

```

```

    | wcslen(DevExt->Name) + 1;
35873|         ULONG LenBytes = sizeof(WCHAR) *
    | LenChars;
35874|
35875|         if ( LenBytes > BufferBytesLeft ) {
35876|             Status =
    | STATUS_BUFFER_OVERFLOW;
35877|             DoneWithSnapShot(p); // don't
    | leave reference count too high
35878|             break;
35879|         }
35880|         wcsncpy ( Buffer, DevExt->Name );
35881|         BufferBytesLeft -= LenBytes;
35882|         Buffer += LenChars;
35883|         Count += LenBytes;
35884|         p =
    | GetNextSnapShotForMaster(&Master->SnapShots,p);
35885|     }
35886|
35887|     if ( NT_SUCCESS(Status) ) {
35888|         if ( BufferBytesLeft >=
    | sizeof(WCHAR) ) {
35889|             // double null terminate
35890|             *Buffer = 0;
35891|             Count += sizeof(WCHAR);
35892|         } else {
35893|             // not enough room to null
    | terminate
35894|             Status =
    | STATUS_BUFFER_OVERFLOW;
35895|         }
35896|     }
35897|
35898|     *Return = Count;
35899| } __finally {
35900|     ReleaseSnapShotForRead();
35901| }
35902| }
35903| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
35904|     Status = GetExceptionCode();
35905|     Debug(DEBUG_DCPSM,("SbGetKernelSnapShotVolumes:
    | Exception %08x\n",Status));
35906| }
35907|
35908| return Status;
35909| }
35910|
35911| //-----
    | -----

```

```

| --
35912|
35913| NTSTATUS SbSetUserName(
35914|         pkSnapshotMaster Master,
35915|         WCHAR *Buffer,
35916|         ULONG BufferSize )
35917| {
35918|     NTSTATUS Status;
35919|
35920|     // Buffer could be invalid
35921|     __try {
35922|         if (
35923|             | AcquireOpenCloseResourceOnly(NULL)==STATUS_WAIT_0 ) {
35924|             __try {
35925|                 Status =
35926|                 | ValidateKernelSnapshotPointer(Master);
35927|                 if ( NT_SUCCESS(Status) ) {
35928|                     ULONG
35929|                     | Amt=min(255*sizeof(WCHAR),BufferSize);
35930|                     ULONG NumBytesInBuffer =
35931|                     | NumBytes(Buffer);
35932|                     ULONG Count =
35933|                     | min(NumBytesInBuffer,Amt);
35934|                     if ( Count==255*sizeof(WCHAR) ) {
35935|                         // hmm, data is larger than we
35936|                         | have a buffer for
35937|                         Status =
35938|                         | STATUS_INVALID_PARAMETER;
35939|                     } else {
35940|                         Status = STATUS_SUCCESS;
35941|                     }
35942|                     | wcsncpy(Master->UserSnapshotName,Buffer,Count);
35943|                     // null terminate
35944|                     | Master->UserSnapshotName[Count]=L'\0';
35945|
35946|                     | Debug(DEBUG_DCPSM,("SbSetUserName: Master=%08x (%s),
35947|                     | Name='%S'\n",
35948|                     Master,
35949|                     (Master->Persistent ?
35950|                     | "persistent" : "temporary"),
35951|                     Master->UserSnapshotName));
35952|
35953|                     // save that the snapshot has
35954|                     | been updated
35955|                     // we need to apply to all
35956|                     | volumes
35957|

```

```

    | PersistentDictionary::BeginUpdate();
35947|         __try {
35948|             GetSnapShotForRead();
35949|         __try {
35950|             pkSnapShotEntry s =
    | GetTopSnapShotForMaster(&Master->SnapShots);
35951|             if ( s ) {
35952|
    | pPersistentDictionary Dict =
    | (pPersistentDictionary)s->Dictionary;
35953|             while ( s ) {
35954|
    | ((pPersistentDictionary)s->Dictionary)->SetSnapShotInfo(
    | Master);
35955|
    | s=GetNextSnapShotForMaster(&Master->SnapShots,s);
35956|             }
35957|         }
35958|     } __finally {
35959|
    | ReleaseSnapShotForRead();
35960|     }
35961| } __finally {
35962|
    | PersistentDictionary::EndUpdate();
35963| }
35964|     LogLinkCreated(Master);
35965| }
35966| }
35967| } __finally {
35968|     ReleaseOpenCloseResource();
35969| }
35970| } else {
35971|     Status = PSM_CANCELED_BY_USER;
35972| }
35973| }

    | __except(ExceptionFilter(GetExceptionInformation())) {
35974|     Status = GetExceptionCode();
35975|     Debug(DEBUG_DCPDM,("SbSetUserName: Exception
    | %08x\n",Status));
35976| }
35977| return Status;
35978| }
35979|
35980| //-----
    | -----
    | --
35981|
35982| NTSTATUS SbGetUserName(
35983|     pkSnapShotMaster Master,

```

```

35984|          WCHAR *Buffer,
35985|          ULONG BufferSize )
35986| {
35987|     NTSTATUS Status;
35988|
35989|     // Buffer could be invalid
35990|     __try {
35991|         Status = ValidateKernelSnapShotPointer(Master);
35992|         if ( NT_SUCCESS(Status) ) {
35993|             if ( BufferSize>sizeof(WCHAR) ) {
35994|                 ULONG NumBytesInUserSnapShotName =
35995|                     | NumBytes(Master->UserSnapShotName);
35996|                 ULONG Count =
35997|                     | min(NumBytesInUserSnapShotName,BufferSize-sizeof(WCHAR))
35998|                     | ;
35999|                 if ( Count==BufferSize ) {
36000|                     // hmm, we have more data than they
36001|                     | gave us a buffer for
36002|                     Status = STATUS_BUFFER_OVERFLOW;
36003|                 } else {
36004|                     Status = STATUS_SUCCESS;
36005|                     | wcsncpy(Buffer,Master->UserSnapShotName,Count);
36006|                     // null terminate
36007|                     Buffer[Count]=L'\0';
36008|                 }
36009|             } else {
36010|                 Status = STATUS_BUFFER_OVERFLOW;
36011|             }
36012|         }
36013|         | __except(ExceptionFilter(GetExceptionInformation())) {
36014|             Status = GetExceptionCode();
36015|             Debug(DEBUG_DCPSM,("SbGetUserName: Exception
36016|             | %08x\n",Status));
36017|         }
36018|         return Status;
36019|     }
36020| }
36021|
36022| //-----
36023| | -----
36024| | --
36025|
36026| NTSTATUS RemoveUserLinks ( pkSnapShotMaster Master )
36027| {
36028|     NTSTATUS Status=STATUS_SUCCESS;
36029|     Debug(DEBUG_DCPSM,("RemoveUserLinks:
36030|     | Master=%08x\n",Master));
36031|
36032|

```

```

36024|  __try {
36025|      Status = ValidateKernelSnapShotPointer(Master);
36026|      if ( NT_SUCCESS(Status) ) {
36027|          ULONG BufferSize=128*1024;
36028|          WCHAR *Buffer =
| (WCHAR*)MemAllocatePoolWithTag(PagedPool,BufferSize,TEMP
| TAG);
36029|          if ( Buffer ) {
36030|              Status = SbGetKernelSnapShotVolumes
| (Master, Buffer, &BufferSize);
36031|              Debug(DEBUG_DCPSM,("RemoveUserLinks:
| SbGetKernelSnapShotVolumes returned Status=%08x,
| BufferSize=%08x\n",Status,BufferSize));
36032|              if ( !Status ) {
36033|                  WCHAR *p=Buffer;
36034|                  WCHAR Temp[512];
36035|
36036|                  while ( *p ) {
36037|                      // remove junction point
| directories for each snapshot.
36038|                      wcsncpy(Temp,p);
36039|                      wcscat(Temp,L"\\");
36040|                      if (
| Master->UserSnapShotName[0]!=0 ) {
36041|                          | wcscat(Temp,Master->UserSnapShotName);
36042|                          | Debug(DEBUG_DCPSM,("RemoveUserLinks: About to call
| SbDeleteMountPoint on '%S'\n",Temp));
36043|                          SbDeleteMountPoint(Temp);
36044|                      }
36045|
36046|                      PDEVICE_OBJECT VD =
| GetVdiskObjectForName( p, Master->Instance);
36047|
36048|                      if ( VD ) {
36049|                          PVDISK_EXTENSION VDE =
| GetVDiskExtension(VD);
36050|                          | swprintf(Temp,L"\\Device\\PsmDevices_%04x\\%s_%d",PSM_LO
| W_COMPATIBLE_VERSION,VDE->Name,VDE->Instance);
36051|
36052|                          // remove any drive letters
| that may have been allocated.
36053|                          | Debug(DEBUG_DCPSM,("RemoveUserLinks: About to call
| CleanupSymLinksForDevice on '%S'\n",Temp));
36054|                          CleanupSymLinksForDevice(
| Temp );
36055|                          Sblo_DismountVolume(Temp);

```

```

36056|         } else {
36057|             Debug(DEBUG_DEVSUP,("Error
| getting vdisk object for '%S' instance
| %d\n",p,Master->Instance));
36058|         }
36059|
36060|         p+=wcslen(p)+1;
36061|     }
36062| }
36063|     FREE_POINTER(Buffer);
36064| } else {
36065|     Debug(DEBUG_DCPSM,("RemoveUserLinks:
| Out of memory\n"));
36066|     Status = STATUS_INSUFFICIENT_RESOURCES;
36067| }
36068| } else {
36069|     Debug(DEBUG_DCPSM,("RemoveUserLinks:
| invalid master snapshot %08x\n",Master));
36070| }
36071| }
| __except(ExceptionFilter(GetExceptionInformation())) {
36072|     Status = GetExceptionCode();
36073|     Debug(DEBUG_DCPSM,("RemoveUserLinks: Exception
| %08x\n",Status));
36074| }
36075| return Status;
36076| }
36077|
36078| //-----
| -----
| --
36079|
36080| NTSTATUS SbGetVolumeCacheInfo( WCHAR *VolumeName,
| pPSM_GetVolumeCacheInfoOut Out )
36081| {
36082|     NTSTATUS Status=STATUS_NOT_FOUND;
36083|     PDEVICE_OBJECT DevObj;
36084|
36085|
36086|     // \device\harddiskvolume1
36087|     DevObj = GetObjectFromName(VolumeName);
36088|     GetStats:
36089|     if ( DevObj ) {
36090|         PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(DevObj);
36091|         // Fix for Granule Size - csd 3/15/01
36092|         ULONG SectorSize = DevExt->BytesPerSector;
36093|         ULONG ulTmp = 0;
36094|
36095|         ulTmp=DevExt->Cache.PSManBitMapSize;

```



```

36096|     ulTmp=(((SECTORS_PER_GRANULE*ulTmp)/1024)/2);
    | // MB
36097|     Out->InitialCacheFileSize = ulTmp;
36098|
36099|     ulTmp=DevExt->Cache.PSManBitMapMaxSize;
36100|     ulTmp=(((SECTORS_PER_GRANULE*ulTmp)/1024)/2);
    | // MB
36101|     Out->MaximumCacheFileSize = ulTmp;
36102|
36103|     ulTmp=DevExt->Cache.CurrentCacheFileSize;
36104|     ulTmp=(((SECTORS_PER_GRANULE*ulTmp))/2); // KB
36105|     Out->CurrentCacheFileSize = ulTmp;
36106|     Status = STATUS_SUCCESS;
36107| } else {
36108|     //
    | \device\PsmDevices_0200\_device_harddiskvolume1
36109|     DevObj = GetObjectFromVDiskName(VolumeName);
36110|     if ( DevObj ) {
36111|         PVDISK_EXTENSION DevExt =
    | GetVDiskExtension(DevObj);
36112|         DevObj=DevExt->PSMDevice;
36113|         if ( DevObj ) {
36114|             goto GetStats;
36115|         }
36116|     }
36117| }
36118| return Status;
36119| }
36120|
36121| //-----
    | -----
    | --
36122|
36123| void GetCacheSizes( PFILTERED_EXTENSION DevExt, ULONG
    | &InitialSize, ULONG &MaxSize );
36124|
36125| //-----
    | -----
    | --
36126|
36127| NTSTATUS PsmCreateFiles( tOpenTransactionInternal
    | *In, PKEVENT AbortEvent )
36128| {
36129|     ULONG i;
36130|     NTSTATUS Status=STATUS_SUCCESS;
36131|     PDEVICE_OBJECT DevObj;
36132|
36133|     if (
    | AcquireOpenCloseResourceOnly(AbortEvent)==STATUS_WAIT_0
    | ) {

```



```

    | STATUS_OBJECT_NAME_NOT_FOUND) || (Status ==
    | STATUS_OBJECT_PATH_NOT_FOUND) ) {
36167|                // this is a new
    | one
36168|                // Params are in
    | MB, convert to number of granules
36169|                ULONG ISC=0;
36170|                ULONG MSC=0;
36171|
36172|                GetCacheSizes(
    | DevExt, ISC, MSC );
36173|
36174|
    | DevExt->Cache.PSManBitMapSize =
    | ISC*((1024*1024)/GRANULE_SIZE);
36175|
    | DevExt->Cache.PSManBitMapMaxSize =
    | MSC*((1024*1024)/GRANULE_SIZE);
36176|
36177|                // no need to do
    | directio since no files exist.
36178|                DevExt->DoDirectIO
    | = FALSE;
36179|                Status =
    | PersistentDictionary::CreateFilesForVolume(DevObj,AbortE
    | vent);
36180|                if (
    | !NT_SUCCESS(Status) ) {
36181|
    | Debug(DEBUG_DCPSM,("CreateFiles: Error %08x creating
    | files for %08x\n",Status,DevObj));
36182|                }
36183|                } else {
36184|
    | Debug(DEBUG_DCPSM,("CreateFiles: Error %08x opening
    | files for %08x\n",Status,DevObj));
36185|                }
36186|                } else {
36187|
    | Debug(DEBUG_DCPSM,("CreateFiles: Error %08x getting
    | state for %08x\n",Status,DevObj));
36188|                }
36189|                } else {
36190|                // cant do anything with
    | cache sizes if snapshots are active
36191|                // go on to next volume
36192|                Status = 0;
36193|                }
36194|        } else { // dev
36195|                // odd, since we passed the

```

```

    | first test
36196|          ASSERT(FALSE);
36197|          Status =
    | STATUS_OBJECT_NAME_NOT_FOUND;
36198|          }
36199|
36200|          if ( !NT_SUCCESS(Status) ) {
36201|              break;
36202|          }
36203|          } // for
36204|      } else {
36205|          // one or more of the passed in objects
    | is wrong
36206|      }
36207|      } __finally {
36208|          ReleaseOpenCloseResource();
36209|      }
36210|  } else {
36211|      Status = PSM_CANCELED_BY_USER;
36212|  }
36213|
36214|  UpdateGlobalStatus (PSM_IDLE); //
    | pd::CreateFilesForVolume sets to PSM_CREATING_FILES;
    | this sets it back.
36215|  return Status;
36216| }
36217|
36218| //-----
    | -----
    | --
36219|
36220| /*--- end of file dcpsm.cpp ---*/
36221|
36222|
36223|
36224| File Listing: DCPSM.h
36225|
36226| //PDEVICE_OBJECT GetObjectFromDriveAndPart ( ULONG
    | DiskNum, ULONG PartNum );
36227| //PDEVICE_OBJECT GetObjectFromVirtualDriveAndPart (
    | ULONG DiskNum, ULONG PartNum );
36228| PDEVICE_OBJECT GetObjectFromName ( WCHAR *Name );
36229| PDEVICE_OBJECT GetObjectFromVDiskName( WCHAR *Name );
36230| PDEVICE_OBJECT GetObjectFromWin32Name( WCHAR *Name );
36231|
36232| #if _WIN32_WINNT < 0x0500
36233| PDEVICE_OBJECT GetObjectFromVolumeNumber( ULONG
    | VolumeNumber );
36234| PDEVICE_OBJECT GetObjectFromDriveLetter( WCHAR
    | DriveLetter );

```

```

36235| #endif
36236|
36237| NTSTATUS SbRunInRing0( PUSER_MODE_ROUTINE UserRoutine,
    | PVOID UserContext );
36238|
36239| NTSTATUS SbOpenPSM( pOT_USER User,
    | tOpenTransactionInInternal *Buffer, ULONG OTOSize,
    | tOpenTransactionOutInternal *OutBuffer, PKEVENT
    | AbortEvent=NULL );
36240|
36241| NTSTATUS
36242| SbPreInit(
36243|     pOT_USER User,
36244|     tOpenTransactionInInternal *Buffer,
36245|     PKEVENT AbortEvent = NULL
36246| );
36247|
36248| // called from unload/shutdown, etc...
36249| NTSTATUS InternalClosePSM( pOT_USER
    | User,pkSnapshotMaster Snapshot );
36250|
36251| NTSTATUS
36252| SbClosePSM(tClosePSMInternal *Buffer);
36253|
36254| NTSTATUS
36255| SbInternalRevertBuffer(
36256|     PDEVICE_OBJECT DeviceObject,
36257|     PVDISK_EXTENSION VDiskExt,
36258|     ULARGE_INTEGER Sector,
36259|     ULONG Count,
36260|     ULONG Delete,
36261|     char *Buffer,
36262|     ULONG *SectorsChanged );
36263|
36264| void PsmOff(void);
36265| void SbOpenPsmThread( pOpenPsmThread Thread );
36266| NTSTATUS SbOpenExclusive(pkSnapshotMaster Snapshot);
36267| NTSTATUS SbCloseExclusive(pkSnapshotMaster Snapshot);
36268|
36269| NTSTATUS SbFreeVolume( pPSM_FreeVolume Volume );
36270| NTSTATUS SbFreeRanges( pPSM_FreeRanges Ranges );
36271| NTSTATUS SbGetProgress( pkSnapshotMaster
    | MasterSnapshot, tPSM_GetProgressOut *Progress );
36272| //int GetLastDeviceNum(void);
36273| NTSTATUS FreeAllSnapShotsForVolume ( PDEVICE_OBJECT
    | DeviceObject );
36274| void MarkAllSnapShotsWithError( pkSnapshotEntry
    | Snapshot, NTSTATUS Error );
36275| void MarkAllSnapShotsWithErrorForVolume(
    | pkSnapshotEntry Snapshot, NTSTATUS Error );

```

```

36276| NTSTATUS IsInUserList( pOT_USER User, pkSnapShotEntry
    | SnapShot);
36277|
36278| NTSTATUS SbGetPersistentSnapShots (
    | tPSM_GetPersistentSnapShotsOut *Out );
36279| NTSTATUS SbGetKernelSnapShotInfo( pkSnapShotMaster
    | Master, tPSM_GetKernelSnapShotInfoOut *Out );
36280| PDEVICE_OBJECT GetVolumeObjectFromVolumeId( ULONG
    | VolumeId );
36281| NTSTATUS SbPreDelInit(void);
36282| NTSTATUS SbGetKernelSnapShotVolumes( pkSnapShotMaster
    | Master, WCHAR *Buffer, ULONG *Return );
36283| NTSTATUS SbGetUserName(
36284|     pkSnapShotMaster Master,
36285|     WCHAR *Buffer,
36286|     ULONG BufferSize );
36287| NTSTATUS SbSetUserName(
36288|     pkSnapShotMaster Master,
36289|     WCHAR *Buffer,
36290|     ULONG BufferSize );
36291| NTSTATUS RemoveUserLinks(
36292|     pkSnapShotMaster Master
36293|     );
36294| PDEVICE_OBJECT GetVdiskObjectForName( WCHAR *Name,
    | ULONG Instance );
36295|
36296| NTSTATUS SbSetKernelSnapShotInfo( pkSnapShotMaster
    | Master, tPSM_SetKernelSnapShotInfo *Info );
36297| NTSTATUS SbMakeVirtualWritesPersistent (
    | pkSnapShotMaster Master );
36298| NTSTATUS SbRemoveVirtualWrites( pkSnapShotMaster Master
    | );
36299| NTSTATUS SbRevertToSnapShot (pkSnapShotMaster Master);
36300|
36301| NTSTATUS WaitForQuiescentPeriod (
36302|     pOT_USER          User,
36303|     tOpenTransactionInInternal *Buffer,
36304|     tkSnapShotMaster      **MasterSnapShot,
36305|     PKEVENT              AbortEvent      =
    | NULL);
36306|
36307| void LogOpen(
36308|     tOpenTransactionInInternal *Buffer,
36309|     pkSnapShotMaster          MasterSnapShot );
36310|
36311| void LogClose(
36312|     pkSnapShotMaster MasterSnapShot );
36313|
36314| NTSTATUS FreePSMVolume( pOT_USER User, PDEVICE_OBJECT
    | DeviceObject, tkSnapShotEntry *SnapShot );

```

```

36315| typedef struct sAbortEventList {
36316|     LIST_ENTRY ListEntry;
36317|     PKEVENT AbortEvent;
36318| } tAbortEventList,*pAbortEventList;
36319|
36320| extern LIST_ENTRY CreationListHead;
36321| NTSTATUS CancelCreationOfSnapShots( );
36322| NTSTATUS SbGetVolumeCacheInfo( WCHAR *VolumeName,
    | pPSM_GetVolumeCacheInfoOut Out );
36323| NTSTATUS FreeResourcesForVolume ( PDEVICE_OBJECT
    | DeviceBeingPSMed,tkSnapshotEntry *Snapshot );
36324| NTSTATUS DelDeviceFromList( PLIST_ENTRY List,
    | pkSnapshotEntry Snapshot);
36325| ULONG NumberOfSnapShotsForVolume (PDEVICE_OBJECT
    | DevObj);
36326| ULONG GetSequenceForNewSnapShot();
36327|
36328|
36329|
36330| File Listing: DEVCON.cpp
36331|
36332| #include "precomp.h"
36333|
36334| #ifdef ALLOC_PRAGMA_DO_NOT_DO
36335|     #pragma alloc_text(PAGE, PSMANDeviceControlObject)
36336| #endif
36337|
36338|
36339|
36340|
36341| /*-----
    | -----*/
36342| NTSTATUS
36343| PSMANDeviceControl(
36344|     PDEVICE_OBJECT DeviceObject,
36345|     PIRP Irp
36346| )
36347|
36348| /*++
36349|
36350| Routine Description:
36351|
36352|     If request is for device it is passed to device
    | handler, otherwise
36353|     it must be for sbpsman object
36354|
36355| Arguments:
36356|
36357|     DeviceObject - Context for the activity.
36358|     Irp          - The device control argument block.

```

```

36359|
36360| Return Value:
36361|
36362|     Status is returned.
36363|
36364| --*/
36365|
36366| {
36367|     NTSTATUS Status;
36368|     PAGED_CODE();
36369|
36370|     switch ( PsmGetObjectType(DeviceObject) ) {
36371|         case OBJECT_INTERNAL :
36372|             Status =
36373|                 | PSMANDeviceControlObject(DeviceObject, Irp);
36374|             break;
36375|         case OBJECT_FILTEREDDISK :
36376|             Status =
36377|                 | PSMANDeviceControlDevice(DeviceObject, Irp);
36378|             break;
36379|         case OBJECT_VIRTUALDISK :
36380|             Status =
36381|                 | PSMANDeviceControlVDisk(DeviceObject, Irp);
36382|             break;
36383|         case OBJECT_FS_OBJECT :
36384|             Status =
36385|                 | PSMANDeviceControlFSObject(DeviceObject, Irp);
36386|             break;
36387|         case OBJECT_FS_FILTER :
36388|             Status =
36389|                 | PSMANDeviceControlFSFilter(DeviceObject, Irp);
36390|             break;
36391|         default:
36392|             Irp->IoStatus.Status = Status =
36393|                 | STATUS_NO_SUCH_DEVICE;
36394|             Irp->IoStatus.Information = 0 ;
36395|             IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
36396|             break;
36397|     }
36398|     return Status;
36399| }
36400| NTSTATUS InitiatePart2OfRebuild()
36401| {
36402|     NTSTATUS Status=STATUS_SUCCESS;
36403|     PDEVICE_OBJECT DevObj =
36404|         | PSMANDriverObject->DeviceObject;
36405|
36406|     Debug(DEBUG_DEVCON,("InitiatePart2OfRebuild: System
36407|         | ready!\n"));

```



```

36401| while ( DevObj ) {
36402|     // if a filtered disk
36403|     if (
36404|         | PsmGetObjectype(DevObj)==OBJECT_FILTEREDDISK ) {
36405|             if ( PersistentDictionary::QueryResetPsm()
36406|                 | ) {
36407|                 NTSTATUS StateReloadStatus =
36408|                 | PersistentDictionary::GetStateForVolume (DevObj);
36409|                 ASSERT (StateReloadStatus ==
36410|                     | STATUS_SUCCESS);
36411|                 | Debug(DEBUG_DICT,("InitiatePart2OfRebuild: ResetPsm ->
36412|                     | calling DeletePsmFiles()\n"));
36413|                 | PersistentDictionary::InitializeFileNamesForVolume
36414|                 | (DevObj);
36415|                 PFILTERED_EXTENSION DevExt =
36416|                 | GetFilteredExtension (DevObj);
36417|                 DeletePsmFilesOnVolume (DevExt);
36418|                 Rebuild_DeleteJunctionsOnVolume
36419|                 | (DevObj);
36420|             }
36421|             HANDLE TempHandle = INVALID_HANDLE_VALUE;
36422|             | pmStartThread(PersistentDictionary::Part2OfRebuildForVol
36423|                 | ume,DevObj,&TempHandle);
36424|             | ZwWaitForSingleObject(TempHandle,FALSE,NULL);
36425|             ZwClose(TempHandle); // dont need the
36426|             | handle anymore.
36427|         } // if filtered disk
36428|     }
36429|     DevObj = DevObj->NextDevice;
36430| } // while(DevObj)
36431| PersistentDictionary::DisableResetPsm();
36432| PersistentDictionary::DisableNoPsm();
36433| return Status;
36434| }
36435| NTSTATUS ExtendAFile();
36436| void DoLink ( void *Arg )
36437| {
36438|     pPSM_SetWin32Link Link=(pPSM_SetWin32Link)Arg;
36439|     NTSTATUS Status;
36440|     UNICODE_STRING deviceNameUnicodeString={0};
36441|     UNICODE_STRING deviceLinkUnicodeString={0};

```

```

36437|
36438|     if(Link->Operation==LINK_SetLink) {
36439|         RtlInitUnicodeString(
36440|             | &deviceLinkUnicodeString,Link->Win32Link );
36441|         RtlInitUnicodeString(
36442|             | &deviceNameUnicodeString,Link->NTDeviceName );
36443|         Status = IoCreateSymbolicLink
36444|             | (&deviceLinkUnicodeString, &deviceNameUnicodeString);
36445|     } else
36446|     if(Link->Operation==LINK_DeleteLink) {
36447|         RtlInitUnicodeString (
36448|             | &deviceLinkUnicodeString, Link->Win32Link );
36449|         IoDeleteSymbolicLink ( &deviceLinkUnicodeString
36450|             | );
36451|         Status = STATUS_SUCCESS;
36452|     } else {
36453|         Status = STATUS_INVALID_PARAMETER;
36454|     }
36455|     Link->Operation = (ULONG)Status;
36456|     return;
36457| }
36458|
36459| LIST_ENTRY
36460|     | PSMEventListHead={&PSMEventListHead,&PSMEventListHead};
36461| LIST_ENTRY
36462|     | PSMNoWaitersListHead={&PSMNoWaitersListHead,&PSMNoWaiter
36463|     | sListHead};
36464|
36465|
36466| NTSTATUS NotifyUserModeOfVolumeOnlineEvent(
36467|     | PFILTERED_EXTENSION FiltExt )
36468| {
36469|     PLIST_ENTRY ListEntry;
36470|     ULONG Err;
36471|     pPSM_GetPSMEventEntry Entry;
36472|
36473|     PAGED_CODE();
36474|     Debug(DEBUG_DEVCON,("DEVCON:
36475|         | NotifyUserModeOfVolumeOnlineEvent
36476|         | %08x\n",FiltExt->DeviceObject));
36477|
36478|     pmAcquireMutex ( &PSMUserMutex, NULL );
36479|
36480|     ListEntry = RemoveHeadList(&PSMEventListHead);
36481|
36482|     if(ListEntry!=&PSMEventListHead) {
36483|         Debug(DEBUG_DEVCON,("DEVCON:
36484|             | NotifyUserModeOfVolumeOnlineEvent: Got event
36485|             | entry\n"));

```

```

36474|    // we have a waiter, lets give it to him
36475|    pPSM_GetPSMEventEntry
    | Entry=CONTAINING_RECORD(ListEntry,tPSM_GetPSMEventEntry,
    | ListEntry);
36476|
36477|    Entry->Irp->IoStatus.Status = STATUS_SUCCESS;
36478|    Entry->Irp->IoStatus.Information =
    | sizeof(tPSM_GetPSMEvent);
36479|    Entry->Event->KernelSnapShotPointer = NULL;
36480|    Entry->Event->Event = PSM_EVENT_VOLUME_ONLINE;
36481|
    | wcscpy(Entry->Event->VolumeGuid,FiltExt->VolumeGuid);
36482|
    | wcscpy(Entry->Event->Uniqueld,FiltExt->Uniqueld);
36483|    wcscpy(Entry->Event->NTName,FiltExt->Name );
36484|
36485|    IoSetCancelRoutine(Entry->Irp,NULL);
36486|    IoCompleteRequest(Entry->Irp,IO_NO_INCREMENT);
36487|    MemFreePool(Entry);
36488|
36489|    Err = STATUS_SUCCESS;
36490| } else {
36491|    Debug(DEBUG_DEVCON,("DEVCON:
    | NotifyUserModeOfVolumeOnlineEvent: no waiters\n"));
36492|    // no waiters, lets queue it up
36493|
    | Entry=(pPSM_GetPSMEventEntry)MemAllocatePoolWithTag(Page
    | dPool,sizeof(tPSM_GetPSMEventEntry),PSM_EVENT_ENTRY_TAG)
    | ;
36494|    if(Entry) {
36495|        Entry->Event =
    | (pPSM_GetPSMEvent)MemAllocatePoolWithTag(PagedPool,sizeo
    | f(tPSM_GetPSMEvent),PSM_EVENT_TAG);
36496|        if(Entry->Event) {
36497|            Entry->Event->KernelSnapShotPointer =
    | NULL;
36498|            Entry->Event->Event =
    | PSM_EVENT_VOLUME_ONLINE;
36499|
    | wcscpy(Entry->Event->VolumeGuid,FiltExt->VolumeGuid);
36500|
    | wcscpy(Entry->Event->Uniqueld,FiltExt->Uniqueld);
36501|
    | wcscpy(Entry->Event->NTName,FiltExt->Name );
36502|
36503|    | InsertTailList(&PSMNoWaitersListHead,&Entry->ListEntry);
36504|        Err = STATUS_SUCCESS;
36505|    } else {
36506|        MemFreePool(Entry);

```

```

36507|         Err = STATUS_INSUFFICIENT_RESOURCES;
36508|     }
36509| } else {
36510|     Err = STATUS_INSUFFICIENT_RESOURCES;
36511| }
36512| }
36513| pmReleaseMutex ( &PSMUserMutex);
36514| return Err;
36515| }
36516|
36517|
36518| NTSTATUS NotifyUserModeOfRegChangeEvent(
    | PFILTERED_EXTENSION FiltExt )
36519| {
36520|     PLIST_ENTRY ListEntry;
36521|     ULONG Err;
36522|     pPSM_GetPSMEventEntry Entry;
36523|
36524|     PAGED_CODE();
36525|     Debug(DEBUG_DEVCON,("DEVCON:
    | NotifyUserModeOfRegChangeEvent
    | %08x\n",FiltExt->DeviceObject));
36526|
36527|     pmAcquireMutex ( &PSMUserMutex, NULL );
36528|
36529|     ListEntry = RemoveHeadList(&PSMEventListHead);
36530|
36531|     if(ListEntry!=&PSMEventListHead) {
36532|         Debug(DEBUG_DEVCON,("DEVCON:
    | NotifyUserModeOfRegChangeEvent: Got Event entry \n"));
36533|         // we have a waiter, lets give it to him
36534|
    | Entry=CONTAINING_RECORD(ListEntry,tPSM_GetPSMEventEntry,
    | ListEntry);
36535|
36536|     Entry->Irp->IoStatus.Status = STATUS_SUCCESS;
36537|     Entry->Irp->IoStatus.Information =
    | sizeof(tPSM_GetPSMEvent);
36538|     Entry->Event->KernelSnapShotPointer = NULL;
36539|     Entry->Event->Event =
    | PSM_EVENT_CLEAN_CLUSTER_REGISTRY;
36540|
    | wcscopy(Entry->Event->VolumeGuid,FiltExt->VolumeGuid);
36541|
    | wcscopy(Entry->Event->UniqueId,FiltExt->UniqueId);
36542|     wcscopy(Entry->Event->NTName,FiltExt->Name );
36543|
36544|     IoSetCancelRoutine(Entry->Irp,NULL);
36545|     IoCompleteRequest(Entry->Irp,IO_NO_INCREMENT);
36546|     MemFreePool(Entry);

```

```

36547|
36548|     Err = STATUS_SUCCESS;
36549| } else {
36550|     Debug(DEBUG_DEVCON,("DEVCON:
    | NotifyUserModeOfRegChangeEvent: No waiters\n"));
36551|     // no waiters, lets queue it up
36552|
    | Entry=(pPSM_GetPSMEventEntry)MemAllocatePoolWithTag(Page
    | dPool,sizeof(tPSM_GetPSMEventEntry),PSM_EVENT_ENTRY_TAG)
    | ;
36553|     if(Entry) {
36554|         Entry->Event =
    | (pPSM_GetPSMEvent)MemAllocatePoolWithTag(PagedPool,sizeo
    | f(tPSM_GetPSMEvent),PSM_EVENT_TAG);
36555|         if(Entry->Event) {
36556|             Entry->Event->KernelSnapShotPointer =
    | NULL;
36557|             Entry->Event->Event =
    | PSM_EVENT_CLEAN_CLUSTER_REGISTRY;
36558|             | wcscpy(Entry->Event->VolumeGuid,FiltExt->VolumeGuid);
36559|             | wcscpy(Entry->Event->UniqueId,FiltExt->UniqueId);
36560|             | wcscpy(Entry->Event->NTName,FiltExt->Name );
36561|
36562|             | InsertTailList(&PSMNoWaitersListHead,&Entry->ListEntry);
36563|             Err = STATUS_SUCCESS;
36564|         } else {
36565|             MemFreePool(Entry);
36566|             Err = STATUS_INSUFFICIENT_RESOURCES;
36567|         }
36568|     } else {
36569|         Err = STATUS_INSUFFICIENT_RESOURCES;
36570|     }
36571| }
36572| pmReleaseMutex ( &PSMUserMutex);
36573| return Err;
36574| }
36575|
36576| VOID
36577| EventCancelRoutine(
36578|     IN PDEVICE_OBJECT DeviceObject,
36579|     IN PIRP Irp
36580| )
36581| {
36582|     PLIST_ENTRY ListEntry;
36583|     BOOLEAN Found=FALSE;
36584|

```

```

36585|   Debug(DEBUG_DEVCON,("DEVCON: Cancel:
      | Irql=%08x\n",KeGetCurrentIrql()));
36586|   // This routine runs at DISPATCH_LEVEL
36587|
36588|   // We do not the cancel spinlock because we do not
      | use
36589|   // the start io method of syncing
36590|   IoReleaseCancelSpinLock(Irp->CancelIrql);
36591|
36592|   Debug(DEBUG_DEVCON,("DEVCON: Cancel2:
      | Irql=%08x\n",KeGetCurrentIrql()));
36593|
36594|   ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
36595|
36596|   pmAcquireMutex ( &PSMUserMutex, NULL );
36597|   ListEntry = PSMEventListHead.Flink;
36598|   while(ListEntry!=&PSMEventListHead) {
36599|       pPSM_GetPSMEventEntry
      | Entry=CONTAINING_RECORD(ListEntry,tPSM_GetPSMEventEntry,
      | ListEntry);
36600|       if(Entry->Irp == Irp) {
36601|           RemoveEntryList(&Entry->ListEntry);
36602|           Found = TRUE;
36603|           break;
36604|       }
36605|       ListEntry=ListEntry->Flink;
36606|   }
36607|   pmReleaseMutex ( &PSMUserMutex);
36608|
36609|   if(Found) {
36610|       IoSetCancelRoutine(Irp,NULL);
36611|       Irp->IoStatus.Status = STATUS_CANCELLED;
36612|       Irp->IoStatus.Information = 0;
36613|       IoCompleteRequest(Irp,IO_NO_INCREMENT);
36614|   }
36615| }
36616|
36617| STATIC NTSTATUS
36618| PSMDeviceControlObject(
36619|     PDEVICE_OBJECT DeviceObject,
36620|     PIRP Irp
36621| )
36622| /*++
36623|
36624| Routine Description:
36625|
36626| This function handles the DeviceIoControl IRP's
36627| that we will be receiving from the executive to
      | control our
36628| device. We will be handing 3 different custom

```

```

| private
36629|   IOCTL's.
36630|   IOCTL_HOLD_REQUEST - To put a request in the queue
| for later
36631|           release.
36632|   IOCTL_RELEASE_REQUEST - To release a request in the
| queue.
36633|   IOCTL_DO_NOTHING - Just to show that we are async.
36634|
36635| Arguments:
36636|
36637|   DriverObject - Pointer to device object for this
| operation
36638|   Irp          - IRP to work on.
36639|
36640| Return Value:
36641|
36642|   NT Status
36643|
36644| --*/
36645| {
36646|   PIO_STACK_LOCATION plrpStack =
| IoGetCurrentIrpStackLocation (Irp) ;
36647|   NTSTATUS ntStatus = 0;
36648|   BOOLEAN CompleteRequest=TRUE;
36649|
36650|   NOT_REFERENCED(DeviceObject);
36651|
36652|   Debug (DEBUG_PROCCALL,("PSManDeviceControlObject
| Called\n")); ;
36653|
36654|   __try {
36655|       Irp->IoStatus.Information = 0 ;
36656|       switch (
| plrpStack->Parameters.DeviceIoControl.IoControlCode ) {
36657| #ifdef DEBUG
36658|         case IOCTL_TEST_FUNCTION : {
36659| #if 0
36660|             Irp->IoStatus.Status = ExtendAFile();
36661| #else
36662|             typedef struct sTest {
36663|                 PDEVICE_OBJECT Volume;
36664|                 ULONG Action;
36665|             } tTest,*pTest;
36666|             pTest
| Test=(pTest)Irp->AssociatedIrp.SystemBuffer;
36667|             HANDLE TempHandle =
| INVALID_HANDLE_VALUE;
36668|             switch(Test->Action) {
36669|                 case 0 :

```

```

36670|                Irp->IoStatus.Status =
    | PersistentDictionary::LoadSnapShotsForVolume(Test->Volum
    | e,TRUE,NULL);
36671|                break;
36672|                case 1 :
36673|                Irp->IoStatus.Status =
    | PersistentDictionary::LoadSnapShotsForVolume(Test->Volum
    | e,FALSE,NULL);
36674|                break;
36675|                case 2 :
36676|                Irp->IoStatus.Status =
    | PersistentDictionary::UnloadSnapShotsForVolume(Test->Vol
    | ume,FALSE);
36677|                break;
36678|                case 3 :
36679|                Irp->IoStatus.Status =
    | PersistentDictionary::MiniUnloadSnapShotsForVolume(Test-
    | >Volume,FALSE);
36680|                break;
36681|                case 4 :
36682|                Irp->IoStatus.Status =
    | STATUS_SUCCESS;
36683|                | pmStartThread(PersistentDictionary::Part2OfRebuildForVol
    | ume,Test->Volume,&TempHandle);
36684|                | ZwWaitForSingleObject(TempHandle,FALSE,NULL);
36685|                ZwClose(TempHandle);
36686|                break;
36687|
36688|                case 5 : {
36689|                Irp->IoStatus.Status =
    | STATUS_SUCCESS;
36690|                | pmStartThread(TestVirginMap,Test->Volume,&TempHandle);
36691|                | ZwWaitForSingleObject(TempHandle,FALSE,NULL);
36692|                ZwClose(TempHandle);
36693|                } break;
36694|
36695|                case 6 : {
36696|                Irp->IoStatus.Status =
    | STATUS_SUCCESS;
36697|                | pmStartThread(DumpProfileInfo_Thread,NULL,&TempHandle);
36698|                | ZwWaitForSingleObject(TempHandle,FALSE,NULL);
36699|                ZwClose(TempHandle);
36700|                } break;
36701|

```



```

36702|                default :
36703|                    Irp->IoStatus.Status =
36704|                        | STATUS_SUCCESS;
36705|                    }
36706|                #endif /*DEBUG*/
36707|                break;
36708|            }
36709|            case IOCTL_BUG_CHECK : {
36710|                // force system to bug check, so it
36711|                | can make a memory.dmp file.
36712|                | KeBugCheckEx((ULONG)0xe100ffff,(ULONG)PSManDriverObject,
36713|                | (ULONG)PsmActive,(ULONG)LastErrorStatus,(ULONG)PSManDevi
36714|                | ceControlObject);
36715|                // if we should ever return...
36716|                Irp->IoStatus.Status =
36717|                    | STATUS_SUCCESS;
36718|                break;
36719|            }
36720|        #endif
36721|        #if MEMDBG
36722|        case IOCTL_GET_MEMORY_USAGE : {
36723|            tGetMemoryUsageOut *Buffer=NULL;
36724|
36725|            | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
36726|            | GetMemUsage\n"));
36727|            // METHOD_BUFFERED
36728|            // Irp->AssociatedIrp.SystemBuffer
36729|            | = Input/Output buffer
36730|
36731|            if (
36732|                | plrpStack->Parameters.DeviceIoControl.OutputBufferLength
36733|                | < sizeof(tGetMemoryUsageOut) ) {
36734|                Debug(DEBUG_DEVCON,("Error!
36735|                | IOCTL buffer not big enough\n"));
36736|                Irp->IoStatus.Status =
36737|                    | STATUS_INVALID_BUFFER_SIZE;
36738|                break;
36739|            }
36740|
36741|            Buffer = (pGetMemoryUsageOut)
36742|            | Irp->AssociatedIrp.SystemBuffer;
36743|
36744|            Buffer->MemTotalNonpagedAlloced =
36745|            | MemTotalNonpagedAlloced;
36746|            Buffer->MemTotalPagedAlloced =
36747|            | MemTotalPagedAlloced;
36748|            Buffer->MemMaxNonpagedAlloced =
36749|            | MemMaxNonpagedAlloced;

```

```

36735|           Buffer->MemMaxPagedAlloced    =
      | MemMaxPagedAlloced;
36736|
36737|           Irp->IoStatus.Status =
      | STATUS_SUCCESS;
36738|           Irp->IoStatus.Information =
      | sizeof(tGetMemoryUsageOut);
36739|           break;
36740|       }
36741| #endif
36742|       case IOCTL_INFORM_SYSTEM_READY : {
36743|
      | PersistentDictionary::SetSystemReady();
36744|           Irp->IoStatus.Status =
      | InitiatePart2OfRebuild();
36745|           break;
36746|       }
36747|       case IOCTL_CREATE_FILES : {
36748|           tOpenTransactionInInternal
      | *Buffer=NULL;
36749|           // METHOD_BUFFERED
36750|           // Irp->AssociatedIrp.SystemBuffer
      | = Input/Output buffer
36751|
36752|           | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
      | CreateFiles\n"));
36753|
36754|           if (
      | plrpStack->Parameters.DeviceIoControl.InputBufferLength
      | < sizeof(tOpenTransactionInInternal) ) {
36755|               Debug(DEBUG_DEVCON,("Error!
      | IOCTL buffer not big enough\n"));
36756|               Irp->IoStatus.Status =
      | STATUS_INVALID_BUFFER_SIZE;
36757|               break;
36758|           }
36759|           Buffer =
      | (tOpenTransactionInInternal *)
      | Irp->AssociatedIrp.SystemBuffer;
36760|
36761|           if ( (!Buffer) ||
      | (Buffer->Size!=sizeof(tOpenTransactionInInternal)) ) {
36762|               Debug(DEBUG_DEVCON,("Error!
      | Buffer is NULL or not right size\n"));
36763|               Irp->IoStatus.Status =
      | STATUS_INVALID_PARAMETER;
36764|               break;
36765|           }
36766|

```

```

36767|           PKEVENT AbortEvent=NULL;
36768|
36769|           if (
36770|               | IsValidHandle(Buffer->AbortEvent) ) {
36771|                   // get system wide object from
36772|                   | process specific handle
36773|                   ntStatus =
36774|                   | ObReferenceObjectByHandle(
36775|                   | Buffer->AbortEvent, //IN HANDLE Handle,
36776|                   | EVENT_QUERY_STATE, //IN ACCESS_MASK DesiredAccess,
36777|                   | *ExEventObjectType, //IN POBJECT_TYPE ObjectType
36778|                   | OPTIONAL,
36779|                   | Irp->RequestorMode, //IN KPROCESSOR_MODE AccessMode,
36780|                   | (PVOID *) &AbortEvent, //OUT PVOID *Object,
36781|                   | NULL //OUT POBJECT_HANDLE_INFORMATION HandleInformation
36782|                   | OPTIONAL
36783|               | );
36784|
36785|               if ( !NT_SUCCESS(ntStatus) ) {
36786|                   Debug(DEBUG_DEVCON,("Error
36787|                   | %08x getting object for abort event
36788|                   | %08x\n",ntStatus,Buffer->AbortEvent));
36789|               }
36790|           }
36791|
36792|           Irp->IoStatus.Status =
36793|           | PsmCreateFiles(Buffer,AbortEvent);
36794|
36795|           ObDereferenceObject(AbortEvent);
36796|           break;
36797|       }
36798|   case IOCTL_GET_PSM_EVENT: {
36799|       if (
36800|           | pIrpStack->Parameters.DeviceIoControl.OutputBufferLength
36801|           | >= sizeof(tPSM_GetPSMEvent) ) {
36802|           pPSM_GetPSMEvent
36803|           | Event=(pPSM_GetPSMEvent)Irp->AssociatedIrp.SystemBuffer;
36804|           pPSM_GetPSMEventEntry Entry;
36805|           PLIST_ENTRY ListEntry;
36806|
36807|           pmAcquireMutex ( &PSMUserMutex,
36808|           | NULL );
36809|           ListEntry =

```

```

    | RemoveHeadList(&PSMNoWaitersListHead);
36798|
36799|
    | if(ListEntry!=&PSMNoWaitersListHead) {
36800|         Debug(DEBUG_DEVCON,("DEVCON:
    | IOCTL_GET_PSM_EVENT: Got queud event\n"));
36801|         // we had an event happen when
    | we were not around, lets handle it
36802|
    | Entry=CONTAINING_RECORD(ListEntry,tPSM_GetPSMEventEntry,
    | ListEntry);
36803|
    | RtlCopyMemory(Event,Entry->Event,sizeof(tPSM_GetPSMEvent
    | ));
36804|         MemFreePool(Entry->Event);
36805|         MemFreePool(Entry);
36806|         Irp->IoStatus.Status=0;
36807|
    | Irp->IoStatus.Information=sizeof(tPSM_GetPSMEvent);
36808|     } else {
36809|         Debug(DEBUG_DEVCON,("DEVCON:
    | IOCTL_GET_PSM_EVENT: Adding waiter\n"));
36810|         // none on the queue, queue it
    | up
36811|
    | Entry=(pPSM_GetPSMEventEntry)MemAllocatePoolWithTag(Page
    | dPool,sizeof(tPSM_GetPSMEventEntry),PSM_EVENT_ENTRY_TAG)
    | ;
36812|         if(Entry) {
36813|             Entry->Event = Event;
36814|             Entry->Irp = Irp;
36815|             Irp->IoStatus.Status =
    | STATUS_PENDING;
36816|
    | InsertTailList(&PSMEventListHead,&Entry->ListEntry);
36817|
    | IoSetCancelRoutine(Irp,EventCancelRoutine);
36818|         IoMarkIrpPending(Irp);
36819|         CompleteRequest = FALSE;
36820|     } else {
36821|         Irp->IoStatus.Status =
    | STATUS_INSUFFICIENT_RESOURCES;
36822|     }
36823| }
36824| pmReleaseMutex ( &PSMUserMutex);
36825| } else {
36826|     Debug(DEBUG_DEVCON,("Error! IOCTL
    | buffer not big enough\n"));
36827|     Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;

```

```

36828|         }
36829|
36830|         break;
36831|     }
36832|     case IOCTL_GET_VOLUME_CACHE_INFO: {
36833|         pPSM_GetVolumeCacheInfoIn In =
36834|         | (pPSM_GetVolumeCacheInfoIn)Irp->AssociatedIrp.SystemBuffer;
36835|         | er;
36836|         pPSM_GetVolumeCacheInfoOut Out =
36837|         | (pPSM_GetVolumeCacheInfoOut)Irp->AssociatedIrp.SystemBuffer;
36838|         | fer;
36839|
36840|         Irp->IoStatus.Status =
36841|         | SbGetVolumeCacheInfo(In->VolumeName,Out);
36842|
36843|         if (
36844|         | NT_SUCCESS(Irp->IoStatus.Status) ) {
36845|             Irp->IoStatus.Information =
36846|             | sizeof(*Out);
36847|         }
36848|         break;
36849|     }
36850|     case IOCTL_LOG_EVENT : {
36851|         pPSM_LogEvent
36852|         | Log=(pPSM_LogEvent)Irp->AssociatedIrp.SystemBuffer;
36853|
36854|         /*lint -save -e740 */
36855|
36856|         | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,Log->EventObject,
36857|         | ntId,Log->Status,NULL,0,Log->NumStrings ? Log->Strings
36858|         | : NULL,Log->NumStrings);
36859|
36860|         /*lint -restore */
36861|         Irp->IoStatus.Status =
36862|         | STATUS_SUCCESS;
36863|         break;
36864|     }
36865|     case IOCTL_GET_PERSISTENT_SNAPSHOTS : {
36866|         tPSM_GetPersistentSnapshotsOut
36867|         | *Buffer=NULL;
36868|         // METHOD_BUFFERED
36869|         // Irp->AssociatedIrp.SystemBuffer
36870|         | = Input/Output buffer
36871|
36872|         if (
36873|         | pIrpStack->Parameters.DeviceIoControl.OutputBufferLength
36874|         | < sizeof(tPSM_GetPersistentSnapshotsOut) ) {
36875|             Debug(DEBUG_DEVCON,("Error!
36876|             | IOCTL buffer not big enough\n"));
36877|             Irp->IoStatus.Status =

```

```

    | STATUS_INVALID_BUFFER_SIZE;
36861|         break;
36862|     }
36863|
36864|     Buffer =
    | (tPSM_GetPersistentSnapShotsOut *)
    | Irp->AssociatedIrp.SystemBuffer;
36865|
36866|     Irp->IoStatus.Status =
    | SbGetPersistentSnapShots(Buffer);
36867|     if (
    | NT_SUCCESS(Irp->IoStatus.Status) ) {
36868|         Irp->IoStatus.Information =
    | sizeof(tPSM_GetPersistentSnapShotsOut);
36869|     }
36870|     break;
36871| }
36872| case IOCTL_GET_KERNEL_SNAPSHOT_INFO : {
36873|     tPSM_GetKernelSnapShotInfoIn
    | *In=NULL;
36874|     tPSM_GetKernelSnapShotInfoOut
    | *Out=NULL;
36875|     // METHOD_BUFFERED
36876|     // Irp->AssociatedIrp.SystemBuffer
    | = Input/Output buffer
36877|
36878|     if (
    | plrpStack->Parameters.DeviceIoControl.OutputBufferLength
    | < sizeof(tPSM_GetKernelSnapShotInfoOut) ) {
36879|         Debug(DEBUG_DEVCON,("Error!
    | IOCTL output buffer not big enough\n"));
36880|         Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
36881|         break;
36882|     }
36883|     if (
    | plrpStack->Parameters.DeviceIoControl.InputBufferLength
    | < sizeof(tPSM_GetKernelSnapShotInfoIn) ) {
36884|         Debug(DEBUG_DEVCON,("Error!
    | IOCTL input buffer not big enough\n"));
36885|         Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
36886|         break;
36887|     }
36888|
36889|     In = (tPSM_GetKernelSnapShotInfoIn
    | *) Irp->AssociatedIrp.SystemBuffer;
36890|     Out =
    | (tPSM_GetKernelSnapShotInfoOut *)
    | Irp->AssociatedIrp.SystemBuffer;

```

```

36891|
36892|         Irp->IoStatus.Status =
        | SbGetKernelSnapShotInfo((pkSnapShotMaster)In->KernelSnap
        | ShotPointer,Out);
36893|         if (
        | NT_SUCCESS(Irp->IoStatus.Status) ) {
36894|             Irp->IoStatus.Information =
        | sizeof(tPSM_GetKernelSnapShotInfoOut);
36895|         }
36896|         break;
36897|     }
36898|     case IOCTL_SET_KERNEL_SNAPSHOT_INFO : {
36899|         tPSM_SetKernelSnapShotInfoIn
        | *Info=NULL;
36900|         // METHOD_BUFFERED
36901|         // Irp->AssociatedIrp.SystemBuffer
        | = Input/Output buffer
36902|
36903|         if (
        | plrpStack->Parameters.DeviceIoControl.InputBufferLength
        | < sizeof(tPSM_SetKernelSnapShotInfo) ) {
36904|             Debug(DEBUG_DEVCON,("Error!
        | IOCTL input buffer not big enough\n"));
36905|             Irp->IoStatus.Status =
        | STATUS_INVALID_BUFFER_SIZE;
36906|             break;
36907|         }
36908|
36909|         Info =
        | (tPSM_SetKernelSnapShotInfoIn *)
        | Irp->AssociatedIrp.SystemBuffer;
36910|
36911|         Irp->IoStatus.Status =
        | SbSetKernelSnapShotInfo((pkSnapShotMaster)Info->KernelSn
        | apShotPointer,&Info->Info);
36912|         break;
36913|     }
36914|     case IOCTL_GET_KERNEL_SNAPSHOT_VOLUMES : {
36915|         tPSM_GetKernelSnapShotInfoIn
        | *In=NULL;
36916|         WCHAR *Out=NULL;
36917|         ULONG Return =
        | plrpStack->Parameters.DeviceIoControl.OutputBufferLength
        | ;
36918|
36919|         // METHOD_BUFFERED
36920|         // Irp->AssociatedIrp.SystemBuffer
        | = Input/Output buffer
36921|
36922|         if (

```

```

    | pIrpStack->Parameters.DeviceIoControl.InputBufferLength
    | < sizeof(tPSM_GetKernelSnapShotInfoIn) ) {
36923|         Debug(DEBUG_DEVCON,("Error!
    | IOCTL input buffer not big enough\n"));
36924|         Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
36925|         break;
36926|     }
36927|
36928|     In = (tPSM_GetKernelSnapShotInfoIn
    | *) Irp->AssociatedIrp.SystemBuffer;
36929|     Out = (WCHAR*)
    | Irp->AssociatedIrp.SystemBuffer;
36930|
36931|     Irp->IoStatus.Status =
    | SbGetKernelSnapShotVolumes((pkSnapShotMaster)In->KernelS
    | napShotPointer, Out, &Return);
36932|     if (
    | NT_SUCCESS(Irp->IoStatus.Status) ) {
36933|         Irp->IoStatus.Information =
    | Return;
36934|     }
36935|     break;
36936| }
36937| case IOCTL_REVERT_TO_PRISTINE: {
36938|     // METHOD_BUFFERED
36939|     // Irp->AssociatedIrp.SystemBuffer
    | = Input/Output buffer
36940|     tPSM_SnapShotPointer *KernelPointer
    | = (tPSM_SnapShotPointer
    | *)Irp->AssociatedIrp.SystemBuffer;
36941|     Irp->IoStatus.Status =
    | SbRemoveVirtualWrites((tkSnapShotMaster
    | *)KernelPointer->KernelSnapShotPointer);
36942|     break;
36943| }
36944| case IOCTL_REVERT_TO_SNAPSHOT: {
36945|     pPSM_RevertToSnapShotIn In =
    | (pPSM_RevertToSnapShotIn)Irp->AssociatedIrp.SystemBuffer
    | ;
36946|     ULONG revertUndoSequence = 0;
36947|     Irp->IoStatus.Status =
    | SbRevertToSnapShot_SeparateThread (
36948|         (tkSnapShotMaster *)
    | (In->KernelSnapShotPointer),
36949|         NULL,
    | //volumeDeviceObject==NULL means "revert all volumes in
    | snapshot"
36950|         In->RevertFlags,
36951|         revertUndoSequence );

```



```

36952|         break;
36953|     }
36954|     case IOCTL_GET_USER_NAME: {
36955|         // METHOD_BUFFERED
36956|         // Irp->AssociatedIrp.SystemBuffer
        | = Input/Output buffer
36957|         WCHAR *Buffer =
        | (WCHAR*)Irp->AssociatedIrp.SystemBuffer;
36958|         tPSM_SnapShotPointer *KernelPointer
        | = (tPSM_SnapShotPointer
        | *)Irp->AssociatedIrp.SystemBuffer;
36959|         Irp->IoStatus.Status =
        | SbGetUserName((tkSnapShotMaster
        | *)KernelPointer->KernelSnapShotPointer,Buffer,pIrpStack-
        | >Parameters.DeviceIoControl.OutputBufferLength);
36960|         if ( Irp->IoStatus.Status==0 ) {
36961|             Irp->IoStatus.Information =
        | NumBytes(Buffer)+sizeof(WCHAR);
36962|         }
36963|         break;
36964|     }
36965|     case IOCTL_SET_USER_NAME: {
36966|         // METHOD_BUFFERED
36967|         // Irp->AssociatedIrp.SystemBuffer
        | = Input/Output buffer
36968|         tPSM_SetUserNameIn *Buffer =
        | (tPSM_SetUserNameIn*)Irp->AssociatedIrp.SystemBuffer;
36969|         if (
        | pIrpStack->Parameters.DeviceIoControl.InputBufferLength>
        | =sizeof(tPSM_SetUserNameIn ) ) {
36970|             Irp->IoStatus.Status =
        | SbSetUserName((pkSnapShotMaster)Buffer->KernelSnapShotPo
        | inter,Buffer->Name,sizeof(Buffer->Name));
36971|         } else {
36972|             Debug(DEBUG_DEVCON,("Error!
        | IOCTL input buffer not big enough\n"));
36973|             Irp->IoStatus.Status =
        | STATUS_INVALID_BUFFER_SIZE;
36974|         }
36975|         break;
36976|     }
36977|
36978|     case IOCTL_SET_FLUSH_ROUTINE : {
36979|         pSetFlushRoutine Buffer=NULL;
36980|
36981|         | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
        | SetFlushRoutine\n"));
36982|         // METHOD_BUFFERED
36983|         // Irp->AssociatedIrp.SystemBuffer

```

```

    | = Input/Output buffer
36984|
36985|         Irp->IoStatus.Information = 0;
36986|
36987|         if (
    | plrpStack->Parameters.DeviceIoControl.InputBufferLength
    | < sizeof(tSetFlushRoutine) ) {
36988|             Debug(DEBUG_DEVCON,("Error!
    | IOCTL buffer not big enough\n"));
36989|             Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
36990|             break;
36991|         }
36992|
36993|         Buffer = (tSetFlushRoutine *)
    | Irp->AssociatedIrp.SystemBuffer;
36994|         if ( Buffer->ZwFlushBuffersFile ) {
36995|             Irp->IoStatus.Status =
    | SetFlushRoutine(Buffer->ZwFlushBuffersFile);
36996|         } else {
36997|             Irp->IoStatus.Status =
    | STATUS_INVALID_PARAMETER;
36998|         }
36999|
37000|         break;
37001|     }
37002|     case IOCTL_OPEN_EX : {
37003|         tkSnapshotMaster *Master=NULL;
37004|         // METHOD_BUFFERED
37005|         // Irp->AssociatedIrp.SystemBuffer
    | = Input/Output buffer
37006|
37007|         if (
    | plrpStack->Parameters.DeviceIoControl.InputBufferLength
    | >= sizeof(PVOID) ) {
37008|             Master =
    | *((pkSnapshotMaster*)(Irp->AssociatedIrp.SystemBuffer));
37009|         }
37010|
37011|         | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
    | OpenExclusive %08x\n",Master));
37012|
37013|             Irp->IoStatus.Status =
    | SbOpenExclusive(Master);
37014|             break;
37015|         }
37016|     case IOCTL_CLOSE_EX : {
37017|         tkSnapshotMaster *Master=NULL;
37018|         // METHOD_BUFFERED

```

```

37019|          // Irp->AssociatedIrp.SystemBuffer
    | = Input/Output buffer
37020|          if (
    | pIrpStack->Parameters.DeviceIoControl.InputBufferLength
    | >= sizeof(PVOID) ) {
37021|              Master =
    | *((pkSnapshotMaster*)(Irp->AssociatedIrp.SystemBuffer));
37022|          }
37023|          | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
    | CloseExclusive %08x\n",Master));
37024|
37025|          Irp->IoStatus.Status =
    | SbCloseExclusive(Master);
37026|          break;
37027|      }
37028|      case IOCTL_GET_PROGRESS : {
37029|          tPSM_GetProgressOut *Buffer=NULL;
37030|          tkSnapshotMaster *Master=NULL;
37031|          // METHOD_BUFFERED
37032|          // Irp->AssociatedIrp.SystemBuffer
    | = Input/Output buffer
37033|
37034|
37035|          if (
    | pIrpStack->Parameters.DeviceIoControl.InputBufferLength
    | >= sizeof(PVOID) ) {
37036|              Master =
    | *((pkSnapshotMaster*)(Irp->AssociatedIrp.SystemBuffer));
37037|          }
37038|
37039|          | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
    | GetProgress %08x\n",Master));
37040|
37041|          if (
    | pIrpStack->Parameters.DeviceIoControl.OutputBufferLength
    | < sizeof(tPSM_GetProgressOut) ) {
37042|              Debug(DEBUG_DEVCON,("Error!
    | IOCTL buffer not big enough\n"));
37043|              Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
37044|              break;
37045|          }
37046|
37047|          Buffer = (tPSM_GetProgressOut *)
    | Irp->AssociatedIrp.SystemBuffer;
37048|
37049|          Irp->IoStatus.Status =
    | SbGetProgress(Master,Buffer);

```

```

37050|         if (
| NT_SUCCESS(Irp->IoStatus.Status) ) {
37051|             Irp->IoStatus.Information =
| sizeof(tPSM_GetProgressOut);
37052|         }
37053|         break;
37054|     }
37055|     case IOCTL_FREE_VOLUME : {
37056|         tPSM_FreeVolume *Buffer=NULL;
37057|         | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
| FreeVolumes\n"));
37058|         // METHOD_BUFFERED
37059|         // Irp->AssociatedIrp.SystemBuffer
| = Input/Output buffer
37060|
37061|         if (
| plrpStack->Parameters.DeviceIoControl.InputBufferLength
| < sizeof(tPSM_FreeVolume) ) {
37062|             Debug(DEBUG_DEVCON,("Error!
| IOCTL buffer not big enough\n"));
37063|             Irp->IoStatus.Status =
| STATUS_INVALID_BUFFER_SIZE;
37064|             break;
37065|         }
37066|
37067|         Buffer = (struct sPSM_FreeVolume *)
| Irp->AssociatedIrp.SystemBuffer;
37068|
37069|         Irp->IoStatus.Status =
| SbFreeVolume(Buffer);
37070|         break;
37071|     }
37072|     case IOCTL_GET_VOLUME_STATS : {
37073|         WCHAR *VolumeName = (WCHAR
| *)Irp->AssociatedIrp.SystemBuffer;
37074|         tPSM_GetStatsRecord *Stats =
| (tPSM_GetStatsRecord *)
| Irp->AssociatedIrp.SystemBuffer;
37075|         PDEVICE_OBJECT DevObj;
37076|
37077|         | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
| GetVolumeStats\n"));
37078|         // METHOD_BUFFERED
37079|         // Irp->AssociatedIrp.SystemBuffer
| = Input/Output buffer
37080|
37081|         if (
| plrpStack->Parameters.DeviceIoControl.OutputBufferLength

```

```

    | < sizeof(tPSM_GetStatsRecord) ) {
37082|         Debug(DEBUG_DEVCON,("Error!
    | IOCTL buffer not big enough\n"));
37083|         Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
37084|         break;
37085|     }
37086|
37087|     DevObj =
    | GetObjectFromName(VolumeName);
37088|     if ( DevObj ) {
37089|         PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DevObj);
37090|
37091|         // FIXFIXFIX shortcut for
    | converting to kilobytes
37092|         Stats->NumberOfWrites.QuadPart
    | = DevExt->NumberOfWriteRequests;
37093|
    | Stats->KilobytesWritten.QuadPart      =
    | DevExt->SectorsWritten / 2;
37094|         Stats->NumberOfReads.QuadPart
    | = DevExt->NumberOfReadRequests;
37095|         Stats->KilobytesRead.QuadPart
    | = DevExt->SectorsRead / 2;
37096|
    | Stats->KilobytesSentToCacheFile.QuadPart=
    | DevExt->CacheWrites;
37097|
37098|         Irp->IoStatus.Status =
    | STATUS_SUCCESS;
37099|         Irp->IoStatus.Information =
    | sizeof(tPSM_GetStatsRecord);
37100|     } else {
37101|         DevObj =
    | GetObjectFromVDiskName(VolumeName);
37102|
37103|         if ( DevObj ) {
37104|             PVDISK_EXTENSION DevExt =
    | GetVDiskExtension(DevObj);
37105|
37106|
    | ASSERT(PsmGetObjectype(DevObj) == OBJECT_VIRTUALDISK);
37107|
    | Stats->NumberOfWrites.QuadPart      =
    | DevExt->NumberOfWriteRequests;
37108|
    | Stats->KilobytesWritten.QuadPart      =
    | DevExt->SectorsWritten / 2;
37109|

```

```

    | Stats->NumberOfReads.QuadPart          =
    | DevExt->NumberOfReadRequests;
37110|
    | Stats->KilobytesRead.QuadPart          =
    | DevExt->SectorsRead / 2;
37111|
    | Stats->KilobytesSentToCacheFile.QuadPart=
    | DevExt->CacheWrites;
37112|
37113|         Irp->IoStatus.Status =
    | STATUS_SUCCESS;
37114|         Irp->IoStatus.Information =
    | sizeof(tPSM_GetStatsRecord);
37115|         } else {
37116|         Irp->IoStatus.Status =
    | STATUS_INVALID_PARAMETER;
37117|         }
37118|     }
37119|
37120|         break;
37121|     }
37122|     case IOCTL_FREE_RANGES : {
37123|         tPSM_FreeRanges *Buffer=NULL;
37124|
    | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
    | FreeRanges\n"));
37125|         // METHOD_BUFFERED
37126|         // Irp->AssociatedIrp.SystemBuffer
    | = Input/Output buffer
37127|
37128|         if (
    | pIrpStack->Parameters.DeviceIoControl.InputBufferLength
    | < sizeof(tPSM_FreeRanges) ) {
37129|             Debug(DEBUG_DEVCON,("Error!
    | IOCTL buffer not big enough\n"));
37130|             Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
37131|             break;
37132|         }
37133|
37134|         Buffer = (tPSM_FreeRangesW *)
    | Irp->AssociatedIrp.SystemBuffer;
37135|
37136|         if (
    | pIrpStack->Parameters.DeviceIoControl.InputBufferLength
    | <
    | (Buffer->NumberOfRanges*sizeof(tPSM_RangeList))+sizeof(t
    | PSM_FreeRanges) ) {
37137|             Debug(DEBUG_DEVCON,("Error!
    | IOCTL buffer not big enough\n"));

```

```

37138|             Irp->IoStatus.Status =
| STATUS_INVALID_BUFFER_SIZE;
37139|             break;
37140|         }
37141|
37142|             Irp->IoStatus.Status =
| SbFreeRanges(Buffer);
37143|             break;
37144|         }
37145|         case IOCTL_GET_VOLUME_INFO: {
37146|             tPSMVolumeInfoOut *Out=NULL;
37147|             tPSMVolumeInfoIn *In=NULL;
37148|             PDEVICE_OBJECT DevObj;
37149|
37150|             // METHOD_BUFFERED
37151|             // Irp->AssociatedIrp.SystemBuffer
| = Input/Output buffer
37152|
37153|             | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
| TurnOnPsm\n"));
37154|             if (
| plrpStack->Parameters.DeviceIoControl.InputBufferLength
| < sizeof(tPSMVolumeInfoIn) ) {
37155|                 Debug(DEBUG_DEVCON,("Error!
| IOCTL buffer not big enough\n"));
37156|                 Irp->IoStatus.Status =
| STATUS_INVALID_BUFFER_SIZE;
37157|                 break;
37158|             }
37159|             In = (tPSMVolumeInfoIn *)
| Irp->AssociatedIrp.SystemBuffer;
37160|             Out = (tPSMVolumeInfoOut *)
| Irp->AssociatedIrp.SystemBuffer;
37161|
37162|             if ( (!In) ||
| (In->Size!=sizeof(tPSMVolumeInfoIn)) ) {
37163|                 Debug(DEBUG_DEVCON,("Error!
| Buffer is NULL or not right size\n"));
37164|                 Irp->IoStatus.Status =
| STATUS_INVALID_PARAMETER;
37165|                 break;
37166|             }
37167|
37168|             if (
| plrpStack->Parameters.DeviceIoControl.OutputBufferLength
| < sizeof(tPSMVolumeInfoOut) ) {
37169|                 Debug(DEBUG_DEVCON,("Error!
| IOCTL buffer not big enough\n"));
37170|                 Irp->IoStatus.Status =

```

```

    | STATUS_INVALID_BUFFER_SIZE;
37171|          break;
37172|          }
37173|
37174|          DevObj =
    | GetObjectFromVdiskName(In->VolumeName);
37175|          if ( DevObj ) {
37176|              PVDISK_EXTENSION DevExt =
    | GetVdiskExtension(DevObj);
37177|              Out->Cluster0Offset =
    | DevExt->Cluster0Offset;
37178|              Irp->IoStatus.Status = 0;
37179|              Irp->IoStatus.Information =
    | sizeof(tPSMVolumeInfoOut);
37180|          } else {
37181|              Irp->IoStatus.Status =
    | STATUS_INVALID_PARAMETER;
37182|          }
37183|
37184|          break;
37185|      }
37186|      case IOCTL_TURNON_PSM : {
37187|          tOpenTransactionInInternal
    | *Buffer=NULL;
37188|          tOpenPsmThread *Thread=NULL;
37189|          // METHOD_BUFFERED
37190|          // Irp->AssociatedIrp.SystemBuffer
    | = Input/Output buffer
37191|
37192|          | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
    | TurnOnPsm\n"));
37193|          if (
    | plrpStack->Parameters.DeviceIoControl.InputBufferLength
    | < sizeof(tOpenTransactionInInternal) ) {
37194|              Debug(DEBUG_DEVCON,("Error!
    | IOCTL buffer not big enough\n"));
37195|              Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
37196|              break;
37197|          }
37198|          Buffer =
    | (tOpenTransactionInInternal *)
    | Irp->AssociatedIrp.SystemBuffer;
37199|
37200|          if ( (!Buffer) ||
    | (Buffer->Size!=sizeof(tOpenTransactionInInternal)) ) {
37201|              Debug(DEBUG_DEVCON,("Error!
    | Buffer is NULL or not right size\n"));
37202|              Irp->IoStatus.Status =

```



```

    | STATUS_INVALID_PARAMETER;
37203|         break;
37204|     }
37205|
37206|         if ( (Buffer->InternalFlags &
    | PSM_IFLAG_NEW_SNAPSHOT) &&
37207|
    | (pIrpStack->Parameters.DeviceIoControl.OutputBufferLengt
    | h < sizeof(tOpenTransactionOutInternal)) ) {
37208|             Debug(DEBUG_DEVCON,("Error!
    | IOCTL buffer not big enough\n"));
37209|             Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
37210|             break;
37211|         }
37212|
37213|
37214|             Debug(DEBUG_INFO,("PSMan:
    | PsGetCurrentProcess=%08x "
37215|             | "PsGetCurrentThread=%08x "
37216|             | "IoGetCurrentProcess=%08x "
37217|             | "KeGetCurrentThread=%08x "
37218|             | "User
    | Thread=%08x\n",
37219|             | PsGetCurrentProcess(),
37220|             | PsGetCurrentThread(),
37221|             | IoGetCurrentProcess(),
37222|             | KeGetCurrentThread(),
37223|             | Irp->Tail.Overlay.Thread
37224|             | ));
37225|
37226|             // we are going to do this async
37227|             Thread = (tOpenPsmThread *)
    | MemAllocatePoolWithTag(PagedPool,sizeof(tOpenPsmThread),
    | OPENTHREADTAG);
37228|             if ( Thread ) {
37229|                 Thread->Irp = Irp;
37230|                 Thread->OTI = Buffer;
37231|                 Thread->OTOSize =
    | pIrpStack->Parameters.DeviceIoControl.OutputBufferLength
    | ;
37232|                 Thread->OTISize =

```

```

    | Irp->Parameters.DeviceIoControl.InputBufferLength;
37233|          /*lint -save -e740 */
37234|          Thread->User = FindPSMUser(
    | PsGetCurrentProcess() , PsGetCurrentThread() );
37235|          /*lint -restore */
37236|          if ( Thread->User ) {
37237|              HANDLE TempHandle=NULL;
37238|
37239| #if _WIN32_WINNT < 0x0500
37240| #define EVENT_QUERY_STATE      0x0001
37241| #define EVENT_MODIFY_STATE    0x0002 // winnt
37242| #define EVENT_ALL_ACCESS
    | (STANDARD_RIGHTS_REQUIRED|SYNCHRONIZE|0x3) // winnt
37243| #endif
37244|
37245|          if (
    | IsValidHandle(Buffer->ErrorEvent) ) {
37246|              // get system wide
    | object from process specific handle
37247|              ntStatus =
    | ObReferenceObjectByHandle(
37248|              | Buffer->ErrorEvent, //IN HANDLE Handle,
37249|              | EVENT_MODIFY_STATE, //IN ACCESS_MASK DesiredAccess,
37250|              | *ExEventObjectType, //IN POBJECT_TYPE ObjectType
    | OPTIONAL,
37251|              | Irp->RequestorMode, //IN KPROCESSOR_MODE AccessMode,
37252|              | (PVOID *) &Thread->User->ErrorEvent, //OUT PVOID
    | *Object,
37253|              | NULL //OUT POBJECT_HANDLE_INFORMATION HandleInformation
    | OPTIONAL
37254|              | );
37255|
37256|          if (
    | !NT_SUCCESS(ntStatus) ) {
37257|              | Debug(DEBUG_DEVCON,("Error %08x getting object for
    | error event %08x\n",ntStatus,Buffer->ErrorEvent));
37258|              | Thread->User->ErrorEvent=NULL;
37259|              }
37260|          }
37261|          if (
    | IsValidHandle(Buffer->AbortEvent) ) {

```

```

37262|                // get system wide
| object from process specific handle
37263|                ntStatus =
| ObReferenceObjectByHandle(
37264|
| Buffer->AbortEvent, //IN HANDLE Handle,
37265|
| EVENT_QUERY_STATE, //IN ACCESS_MASK DesiredAccess,
37266|
| *ExEventObjectType, //IN POBJECT_TYPE ObjectType
| OPTIONAL,
37267|
| Irp->RequestorMode, //IN KPROCESSOR_MODE AccessMode,
37268|
| (PVOID *) &Thread->User->AbortEvent, //OUT PVOID
| *Object,
37269|
| NULL //OUT POBJECT_HANDLE_INFORMATION HandleInformation
| OPTIONAL
37270|
| );
37271|
37272|                if (
| INT_SUCCESS(ntStatus) ) {
37273|
| Debug(DEBUG_DEVCON,("Error %08x getting object for
| abort event %08x\n",ntStatus,Buffer->AbortEvent));
37274|
| Thread->User->AbortEvent=NULL;
37275|                }
37276|                }
37277|
37278|                // its pending so do not
| complete the request
37279|                Irp->IoStatus.Status =
| STATUS_PENDING;
37280|                IoMarkIrpPending(Irp);
37281|
37282|                ntStatus = pmStartThread(
37283|
| (PKSTART_ROUTINE)SbOpenPsmThread, // IN
| PKSTART_ROUTINE StartRoutine,
37284|
| (PVOID)Thread,                // IN PVOID
| StartContext
37285|
| &TempHandle                // OUT PHANDLE
| ThreadHandle,
37286|
| );
37287|                if ( NT_SUCCESS(ntStatus) )

```

```

| {
37288|                // dont need the handle
| anymore.
37289|                ZwClose(TempHandle);
37290|                TempHandle=NULL;
37291|
| try_return(CompleteRequest = FALSE);
37292|                } else {
37293|
| Debug(DEBUG_DCPSM,("Error %08x creating
| thread\n",ntStatus));
37294|                Irp->IoStatus.Status =
| ntStatus;
37295|                if (
| Thread->User->AbortEvent ) {
37296|
| ObDereferenceObject(Thread->User->AbortEvent);
37297|                }
37298|                if (
| Thread->User->ErrorEvent ) {
37299|
| ObDereferenceObject(Thread->User->ErrorEvent);
37300|                }
37301|
| Thread->User->AbortEvent = NULL;
37302|
| Thread->User->ErrorEvent = NULL;
37303|                FREE_POINTER(Thread);
37304|                }
37305|                } else {
37306|                Debug(DEBUG_DCPSM,("PSM can
| not find user!!!! failing open %08x
| %08x\n",PsGetCurrentProcess() , PsGetCurrentThread()));
37307|                Irp->IoStatus.Status =
| STATUS_INVALID_HANDLE;
37308|                FREE_POINTER(Thread);
37309|                }
37310|                } else {
37311|                Debug(DEBUG_DCPSM,("Out of
| memory for new thread struct"));
37312|                Irp->IoStatus.Status =
| STATUS_INSUFFICIENT_RESOURCES;
37313|                }
37314|                break;
37315|                }
37316|                case IOCTL_TURNOFF_PSM : {
37317|                tClosePSMInternal *Buffer=NULL;
37318|
| Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
| TurnOffPsm\n"));

```

```

37319|          // METHOD_BUFFERED
37320|          // Irp->AssociatedIrp.SystemBuffer
| = Input/Output buffer
37321|
37322|          if (
| plrpStack->Parameters.DeviceIoControl.InputBufferLength
| < sizeof(tClosePSMInternal) ) {
37323|              Irp->IoStatus.Status =
| SbClosePSM(NULL);
37324|          } else {
37325|              Buffer = (tClosePSMInternal *)
| Irp->AssociatedIrp.SystemBuffer;
37326|              Irp->IoStatus.Status =
| SbClosePSM(Buffer);
37327|          }
37328|          break;
37329|      }
37330|      case IOCTL_GET_VERSION : {
37331|          tPSM_VersionInfo *Buffer=NULL;
37332|
| Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
| GetVersion\n"));
37333|          // METHOD_BUFFERED
37334|          // Irp->AssociatedIrp.SystemBuffer
| = Input/Output buffer
37335|
37336|          if (
| plrpStack->Parameters.DeviceIoControl.InputBufferLength
| < sizeof(DWORD) ) {
37337|              Debug(DEBUG_DEVCON,("Error!
| Input IOCTL buffer not big enough\n"));
37338|              Irp->IoStatus.Status =
| STATUS_INVALID_BUFFER_SIZE;
37339|              break;
37340|          }
37341|
37342|          if (
| plrpStack->Parameters.DeviceIoControl.OutputBufferLength
| < sizeof(tPSM_VersionInfo2) ) {
37343|              Debug(DEBUG_DEVCON,("Error!
| Output IOCTL buffer not big enough\n"));
37344|              Irp->IoStatus.Status =
| STATUS_INVALID_BUFFER_SIZE;
37345|              break;
37346|          }
37347|
37348|          Buffer = (struct _sPSM_VersionInfo2
| *) Irp->AssociatedIrp.SystemBuffer;
37349|
37350|          if ( !Buffer ) {

```

```

37351|             Debug(DEBUG_DEVCON,("Error!
| Buffer is NULL\n"));
37352|             Irp->IoStatus.Status =
| STATUS_INVALID_PARAMETER;
37353|             break;
37354|         }
37355|
37356|         // inbound is a DWORD size
37357|         // we are doing this on the fact
| that the input and output buffers
37358|         // are the same and that Size is
| the first argument of the output.
37359|
37360| #include "buildnum.h"
37361|
37362|         if (
| (Buffer->Size==sizeof(tPSM_VersionInfo1)) ||
| (Buffer->Size==sizeof(tPSM_VersionInfo2)) ) {
37363|             Buffer->Version =
| (_BuildNumber_ << 16) | PSM_CURRENT_VERSION;
37364|             Buffer->LoVersion =
| PSM_LOW_COMPATIBLE_VERSION;
37365|             Buffer->OSType =
| PSM_OS_WIN_NT_SERVER; // FIXFIXFIX
37366|             Buffer->OSVersion =
| 0x04000565; // FIXFIXFIX (nt 4.0 build 1381)
37367|             Buffer->CommunicationMethods =
| 0;
37368|
37369|             Irp->IoStatus.Status =
| 0;
37370|             Irp->IoStatus.Information =
| sizeof(tPSM_VersionInfo);
37371|         } else {
37372|             Irp->IoStatus.Status =
| STATUS_INVALID_BUFFER_SIZE;
37373|         }
37374|         break;
37375|     }
37376|     case IOCTL_GET_ERROR : {
37377|         tPSM_GetErrorOut *Buffer=NULL;
37378|         tkSnapShotMaster *Master=NULL;
37379|
37380|         // METHOD_OUT_DIRECT
37381|         // Irp->MdlAddress = Output buffer
37382|         // Irp->AssociatedIrp.SystemBuffer
| = Input buffer
37383|
37384|         if (
| pIrpStack->Parameters.DeviceIoControl.InputBufferLength

```

```

    | >= sizeof(PVOID) ) {
37385|         Master =
    | *((pkSnapShotMaster*)(Irp->AssociatedIrp.SystemBuffer));
37386|     }
37387|
37388|     | Debug(DEBUG_PROCCALL,("PSManDeviceControlObject:
    | GetError %08x\n",Master));
37389|
37390|
37391|     if (
    | pIrpStack->Parameters.DeviceIoControl.OutputBufferLength
    | < sizeof(tPSM_GetErrorOut) ) {
37392|         Debug(DEBUG_DEVCON,("Error!
    | Output IOCTL buffer not big enough\n"));
37393|         Irp->IoStatus.Status =
    | STATUS_INVALID_BUFFER_SIZE;
37394|         break;
37395|     }
37396|
37397|     /*lint -save -e641 */
37398| #if _WIN32_WINNT >= 0x0500
37399|         Buffer = (tPSM_GetErrorOut *)
    | MmGetSystemAddressForMdlSafe( Irp->MdlAddress,
    | NormalPagePriority );
37400| #else
37401|         Buffer = (tPSM_GetErrorOut *)
    | MmGetSystemAddressForMdl( Irp->MdlAddress );
37402| #endif
37403|     /*lint -restore */
37404|
37405|     if ( !Buffer ) {
37406|         Debug(DEBUG_DEVCON,("Error!
    | Buffer is NULL\n"));
37407|         Irp->IoStatus.Status =
    | STATUS_INVALID_PARAMETER;
37408|         break;
37409|     }
37410|
37411|     if ( (Master) &&
    | (Master->Status!=0) ) {
37412|         // reason snapshot failed
37413|         Buffer->ErrorCode =
    | Master->Status;
37414|     } else {
37415|         // reason psm failed
37416|         Buffer->ErrorCode =
    | LastErrorStatus;
37417|     }
37418|

```

```

37419|           // same here, except it will get
| converted to the win32 error code
37420|           // by the io manager for us. this
| is the reason we use METHOD_OUT_DIRECT
37421|           // is so we can return a error
| code, otherwise, when buffering, the buffer
37422|           // will not be copied into the user
| buffer.
37423|           Irp->IoStatus.Status      =
| Buffer->ErrorCode;
37424|           Irp->IoStatus.Information =
| sizeof(tPSM_GetErrorOut);
37425|           break;
37426|       }
37427|       case IOCTL_SET_WIN32_LINK : {
37428|           pPSM_SetWin32Link
| Link=(pPSM_SetWin32Link)Irp->AssociatedIrp.SystemBuffer;
37429|           NTSTATUS
| Status=STATUS_INVALID_BUFFER_SIZE;
37430|           HANDLE TempHandle;
37431|
37432|           if
| (plrpStack->Parameters.DeviceIoControl.InputBufferLength
| >= sizeof(tPSM_SetWin32Link)) {
37433|               // launch thread in system process
37434|
| PsCreateSystemThread(&TempHandle,THREAD_ALL_ACCESS,NULL,
| NULL,NULL,DoLink,Link);
37435|               // and wait for it to finish
37436|
| ZwWaitForSingleObject(TempHandle,FALSE,NULL);
37437|               ZwClose(TempHandle);
37438|               Status = (NTSTATUS)Link->Operation;
37439|           }
37440|
37441|           Irp->IoStatus.Status      = Status;
37442|           Irp->IoStatus.Information = 0;
37443|           break;
37444|       }
37445|
37446|       default:
37447|           Debug(DEBUG_ERROR,("Error! Unknown
| IOCTL received\n")) ;
37448|           Irp->IoStatus.Status =
| STATUS_NOT_IMPLEMENTED ;
37449|           break;
37450|       }
37451|       try_exit: NOTHING;
37452|   } __except(
| ExceptionFilter(GetExceptionInformation()) ) {

```



```

37453|     Irp->IoStatus.Status = GetExceptionCode();
37454|     Debug(DEBUG_THREAD,("PSManDeviceControlObject:
| Error! Exception %08x\n", Irp->IoStatus.Status));
37455| }
37456|
37457| ntStatus = Irp->IoStatus.Status;
37458| if ( CompleteRequest ) {
37459|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
37460| }
37461|
37462| //Debug (DEBUG_PROCCALL,("PSManDeviceControlObject
| Done\n")) ;
37463| return ntStatus;
37464| } // end PSManDeviceControlObject()
37465|
37466|
37467|
37468| /*-----
| -----*/
37469| STATIC NTSTATUS
37470| PSManDiskGetGeometry(
37471|     IN PDEVICE_OBJECT DeviceObject,
37472|     IN PIRP Irp,
37473|     IN PVOID /*Context*/
37474| )
37475|
37476| /*++
37477|
37478| Routine Description:
37479|
37480| This is the completion routine for
37481| | IOCTL_DISK_GET_GEOMETRY.
37482| Called at <=DISPATCH_LEVEL
37483| Arguments:
37484|
37485| DeviceObject - Pointer to device object to being
37486| | shutdown by system.
37487| Irp - IRP involved.
37488| Context - NULL
37489| Return Value:
37490|
37491| NTSTATUS
37492|
37493| --*/
37494|
37495| {
37496|     //NOT_REFERENCED(Context);
37497|     PFILTERED_EXTENSION DevExt =

```

```

    | GetFilteredExtension(DeviceObject);
37498|
37499|    // if removable, adjust what we know about the
    | device
37500|    if ( Irp->IoStatus.Status==0 ) {
37501|        PDISK_GEOMETRY Geometry =
    | (PDISK_GEOMETRY)Irp->AssociatedIrp.SystemBuffer;
37502|
37503|        if ( Geometry ) {
37504|            PSBPSMAN_EXTENSION SbotExt =
    | (PSBPSMAN_EXTENSION) GetDeviceExtension(PManObject);
37505|
    | ASSERT(SbotExt->ObjectType==OBJECT_INTERNAL);
37506|
37507|            // get the disk geometry
37508|            DevExt->Cylinders      =
    | Geometry->Cylinders;
37509|            DevExt->MediaType      =
    | Geometry->MediaType;
37510|            DevExt->TracksPerCylinder =
    | Geometry->TracksPerCylinder;
37511|            DevExt->SectorsPerTrack =
    | Geometry->SectorsPerTrack;
37512|            DevExt->BytesPerSector  =
    | Geometry->BytesPerSector;
37513|
37514|            // save largest BPS request
37515|            if ( DevExt->BytesPerSector >
    | SbotExt->LargestBPS ) {
37516|                SbotExt->LargestBPS =
    | DevExt->BytesPerSector;
37517|            }
37518|
37519|
37520|            Debug(DEBUG_DEVCON,("DevCon: GetGeomtry:
    | Device=%p, Cyls=%l64d, Heads=%d, SPT=%d, BPS=%d\n",
37521|                DevExt->DeviceObject,
37522|                Geometry->Cylinders,
37523|
    | Geometry->TracksPerCylinder,
37524|
    | Geometry->SectorsPerTrack,
37525|
    | Geometry->BytesPerSector
37526|                ));
37527|
37528|        } else {
37529|            Debug(DEBUG_DEVCON,("Error! buffer
    | NULL\n"));
37530|        }

```

```

37531|    }
37532|
37533|    if ( Irp->PendingReturned ) {
37534| //      Debug(DEBUG_DEVCON,("Marking Irp as
    | pending\n"));
37535|      IoMarkIrpPending(Irp);
37536|    }
37537|
37538| //  Debug(DEBUG_PROCCALL,("PSManDiskGetGeometry
    | Done\n"));
37539|    return STATUS_SUCCESS;
37540| }
37541|
37542| #ifdef DEBUG
37543|
37544| /*-----
    | -----*/
37545| STATIC NTSTATUS
37546| PSManDiskGetDriveLayout(
37547|     IN PDEVICE_OBJECT DeviceObject,
37548|     IN PIRP          Irp,
37549|     IN PVOID          /*Context*/
37550| )
37551|
37552| /*++
37553|
37554| Routine Description:
37555|
37556| This is the completion routine for
    | IOCTL_DISK_GET_GEOMETRY.
37557|
37558| Called at <=DISPATCH_LEVEL
37559| Arguments:
37560|
37561| DeviceObject - Pointer to device object to being
    | shutdown by system.
37562| Irp          - IRP involved.
37563| Context      - NULL
37564|
37565| Return Value:
37566|
37567| NTSTATUS
37568|
37569| --*/
37570|
37571| {
37572|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DeviceObject);
37573|     ULONG partNumber;
37574|

```

```

37575| //NOT_REFERENCED(Context);
37576|
37577| // if removable, adjust what we know about the
    | device
37578| if ( Irp->IoStatus.Status==0 ) {
37579|     PDRIVE_LAYOUT_INFORMATION partitionInfo=
    | (PDRIVE_LAYOUT_INFORMATION)Irp->AssociatedIrp.SystemBuff
    | er;
37580|
37581|     if ( partitionInfo ) {
37582|         Debug(DEBUG_DEVCON,("%d partitions
    | found\n",partitionInfo->PartitionCount));
37583|         for ( partNumber = 0; partNumber <
    | partitionInfo->PartitionCount; partNumber++ ) {
37584|             Debug(DEBUG_DEVCON,("%2d,%2d:
    | Offset=%08x%08x, Length=%08x%08x, Hidden=%08x,
    | Type=%02x\n",
37585|                                     DevExt->DiskNumber,
37586|                                     partNumber,
37587|
    | partitionInfo->PartitionEntry[partNumber].StartingOffset
    | .HighPart,
37588|
    | partitionInfo->PartitionEntry[partNumber].StartingOffset
    | .LowPart,
37589|
    | partitionInfo->PartitionEntry[partNumber].PartitionLengt
    | h.HighPart,
37590|
    | partitionInfo->PartitionEntry[partNumber].PartitionLengt
    | h.LowPart,
37591|
    | partitionInfo->PartitionEntry[partNumber].HiddenSectors,
37592|
    | partitionInfo->PartitionEntry[partNumber].PartitionType
37593|                                     ));
37594|
37595|     }
37596|
37597| } else {
37598|     Debug(DEBUG_DEVCON,("Error! buffer
    | NULL\n"));
37599| }
37600| }
37601|
37602| if ( Irp->PendingReturned ) {
37603| //     Debug(DEBUG_DEVCON,("Marking Irp as
    | pending\n"));
37604|     IoMarkIrpPending(Irp);
37605| }

```

```

37606|
37607| //  Debug(DEBUG_PROCCALL,("PSManDiskGetDriveLayout
    | Done\n"));
37608|  return STATUS_SUCCESS;
37609| }
37610| #endif
37611|
37612| #if _WIN32_WINNT >= 0x0500
37613| /*-----
    | -----*/
37614| STATIC NTSTATUS
37615| PSManDiskGetDeviceNumber(
37616|     IN PDEVICE_OBJECT DeviceObject,
37617|     IN PIRP Irp,
37618|     IN PVOID /*Context*/
37619| )
37620|
37621| /*++
37622|
37623| Routine Description:
37624|
37625| This is the completion routine for
    | IOCTL_DISK_GET_GEOMETRY.
37626|
37627| Called at <=DISPATCH_LEVEL
37628| Arguments:
37629|
37630| DeviceObject - Pointer to device object to being
    | shutdown by system.
37631| Irp - IRP involved.
37632| Context - NULL
37633|
37634| Return Value:
37635|
37636| NTSTATUS
37637|
37638| --*/
37639|
37640| {
37641|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DeviceObject);
37642|
37643|     if ( Irp->IoStatus.Status==0 ) {
37644|         PSTORAGE_DEVICE_NUMBER
    | DevNum=(PSTORAGE_DEVICE_NUMBER)Irp->AssociatedIrp.System
    | Buffer;
37645|         Debug(DEBUG_DEVCON,("Device %X is DT %X device
    | %d partition %d
    | (%S)\n",DeviceObject,DevNum->DeviceType,DevNum->DeviceNu
    | mber,DevNum->PartitionNumber,DevExt->Name));

```

```

37646| } else {
37647|     Debug(DEBUG_DEVCON,("GetDeviceNumber: Error
| %08x\n",Irp->IoStatus.Status));
37648| }
37649|
37650| if ( Irp->PendingReturned ) {
37651|     IoMarkIrpPending(Irp);
37652| }
37653|
37654| return STATUS_SUCCESS;
37655| }
37656|
37657| /*-----
| -----*/
37658| STATIC NTSTATUS
37659| PSMANDiskGetVolumeNumber(
37660|     IN PDEVICE_OBJECT DeviceObject,
37661|     IN PIRP          Irp,
37662|     IN PVOID          /*Context*/
37663| )
37664|
37665| /*++
37666|
37667| Routine Description:
37668|
37669| This is the completion routine for
| IOCTL_DISK_GET_GEOMETRY.
37670|
37671| Called at <=DISPATCH_LEVEL
37672| Arguments:
37673|
37674| DeviceObject - Pointer to device object to being
| shutdown by system.
37675| Irp          - IRP involved.
37676| Context      - NULL
37677|
37678| Return Value:
37679|
37680| NTSTATUS
37681|
37682| --*/
37683|
37684| {
37685|     PFILTERED_EXTENSION DevExt = GetFilteredExtension
| (DeviceObject);
37686|
37687| if ( Irp->IoStatus.Status==0 ) {
37688|     PVOLUME_NUMBER
| VolNum=(PVOLUME_NUMBER)Irp->AssociatedIrp.SystemBuffer;
37689|     Debug(DEBUG_DEVCON,("Volume %X is number %d on

```

```

    | '%-8.8s\n", DeviceObject, VolNum->VolumeNumber, VolNum->Vo
    | lumeManagerName));
37690| } else {
37691|     Debug(DEBUG_DEVCON, ("GetVolumeNumber: Error
    | %08x\n", Irp->IoStatus.Status));
37692| }
37693|
37694| if ( Irp->PendingReturned ) {
37695|     IoMarkIrpPending(Irp);
37696| }
37697|
37698| return STATUS_SUCCESS;
37699| }
37700|
37701| /*-----
    | -----*/
37702| STATIC NTSTATUS
37703| PSMANDiskGetDiskExtents(
37704|     IN PDEVICE_OBJECT DeviceObject,
37705|     IN PIRP Irp,
37706|     IN PVOID /*Context*/
37707| )
37708|
37709| /*++
37710|
37711| Routine Description:
37712|
37713| This is the completion routine for
    | IOCTL_DISK_GET_GEOMETRY.
37714|
37715| Called at <=DISPATCH_LEVEL
37716| Arguments:
37717|
37718| DeviceObject - Pointer to device object to being
    | shutdown by system.
37719| Irp - IRP involved.
37720| Context - NULL
37721|
37722| Return Value:
37723|
37724| NTSTATUS
37725|
37726| --*/
37727|
37728| {
37729|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DeviceObject);
37730|
37731| if ( Irp->IoStatus.Status==0 ) {
37732|     PVOLUME_DISK_EXTENTS

```

```

    | DE=(PVOLUME_DISK_EXTENTS)Irp->AssociatedIrp.SystemBuffer
    | ;
37733|     Debug(DEBUG_DEVCON,("Device %X has %d
    | extents\n",DeviceObject,DE->NumberOfDiskExtents));
37734|     // FIXFIXFIX !FIX!FIX! need to keep what
    | extents there are
37735|     for ( ULONG i=0;i<DE->NumberOfDiskExtents;i++ )
    | {
37736|         Debug(DEBUG_DEVCON,(" %08x: %l64x
    | %l64x\n",DE->Extents[i].DiskNumber,DE->Extents[i].Starti
    | ngOffset,DE->Extents[i].ExtentLength));
37737|     }
37738| } else {
37739|     Debug(DEBUG_DEVCON,("GetDiskExtents: Error
    | %08x\n",Irp->IoStatus.Status));
37740| }
37741|
37742| if ( Irp->PendingReturned ) {
37743|     IoMarkIrpPending(Irp);
37744| }
37745|
37746| return STATUS_SUCCESS;
37747| }
37748|
37749|
37750| typedef struct s12 {
37751|     ULONG Data1;
37752|     ULONG Data2;
37753|     ULONG Data3;
37754| } t12,*p12;
37755|
37756| /*-----
    | -----*/
37757| STATIC NTSTATUS
37758| PSMANDiskQueryUniqueId(
37759|     IN PDEVICE_OBJECT DeviceObject,
37760|     IN PIRP Irp,
37761|     IN PVOID /*Context*/
37762| )
37763|
37764| /*++
37765|
37766| Routine Description:
37767|
37768| This is the completion routine
37769|
37770| Called at <=DISPATCH_LEVEL
37771| Arguments:
37772|
37773| DeviceObject - Pointer to device object

```



```

37774|   Irp      - IRP involved.
37775|   Context   - NULL
37776|
37777| Return Value:
37778|
37779|   NTSTATUS
37780|
37781| --*/
37782|
37783| {
37784|   PFILTERED_EXTENSION DevExt =
37785|   | GetFilteredExtension(DeviceObject);
37786|   if ( Irp->IoStatus.Status==0 ) {
37787|       PMOUNTDEV_UNIQUE_ID UniqueId =
37788|       | (PMOUNTDEV_UNIQUE_ID)Irp->AssociatedIrp.SystemBuffer;
37789|       ULONG Len = LENGTH_OF_UNIQUE;
37790|       p12 p=(p12)&UniqueId->UniqueId;
37791|       BufferToHexWChar( p, UniqueId->UniqueIdLength,
37792|       | DevExt->UniqueId, &Len);
37793|       Debug(DEBUG_DEVCON,("%08x: UniqueId Len=%d,
37794|       | %08x %08x %08x
37795|       | (%S)\n",DeviceObject,UniqueId->UniqueIdLength,
37796|       | p->Data1,p->Data2,p->Data3,DevExt->UniqueId));
37797|   } else {
37798|       Debug(DEBUG_DEVCON,("UniqueId error
37799|       | %08x\n",Irp->IoStatus.Status));
37800|   }
37801|
37802|   if ( Irp->PendingReturned ) {
37803|       IoMarkIrpPending(Irp);
37804|   }
37805|   return STATUS_SUCCESS;
37806| }
37807|
37808| /*-----*/
37809| | -----*/
37810|
37811| STATIC NTSTATUS
37812| PSMANDiskUniqueIdChangeNotify(
37813|     IN PDEVICE_OBJECT
37814|     | DeviceObject,
37815|     IN PIRP          Irp,
37816|     IN PVOID
37817|     | /*Context*/
37818| )
37819|
37820| /*++

```

```

37814|
37815| Routine Description:
37816|
37817|   This is the completion routine
37818|
37819|   Called at <=DISPATCH_LEVEL
37820| Arguments:
37821|
37822|   DeviceObject - Pointer to device object
37823|   Irp          - IRP involved.
37824|   Context      - NULL
37825|
37826| Return Value:
37827|
37828|   NTSTATUS
37829|
37830| --*/
37831|
37832| {
37833|   PFILTERED_EXTENSION DevExt =
37834|   | GetFilteredExtension(DeviceObject);
37835|   __try {
37836|       if ( Irp->IoStatus.Status==0 ) {
37837|           PMOUNTDEV_UNIQUE_ID_CHANGE_NOTIFY_OUTPUT
37838|           | UniqueId =
37839|           | (PMOUNTDEV_UNIQUE_ID_CHANGE_NOTIFY_OUTPUT)Irp->Associate
37840|           | dIrp.SystemBuffer;
37841|           PCHAR B=(PCHAR)
37842|           | Irp->AssociatedIrp.SystemBuffer;
37843|           ULONG Len = LENGTH_OF_UNIQUE;
37844|           p12
37845|           | old=(p12)(B+UniqueId->OldUniqueIdOffset);
37846|           p12
37847|           | newid=(p12)(B+UniqueId->NewUniqueIdOffset);
37848|           BufferToHexWChar( newid,
37849|           | UniqueId->NewUniqueIdLength, DevExt->UniqueId, &Len);
37850|           Debug(DEBUG_DEVCON,("UniqueId Size=%d,
37851|           | OldLen=%d, %08x %08x
37852|           | %08x\n",UniqueId->Size,UniqueId->OldUniqueIdLength,old->
37853|           | Data1,old->Data2,old->Data3));
37854|           Debug(DEBUG_DEVCON,("
37855|           | NewLen=%d, %08x %08x %08x
37856|           | (%S)\n",UniqueId->NewUniqueIdLength,newid->Data1,newid->
37857|           | Data2,newid->Data3,DevExt->UniqueId));
37858|       } else {
37859|           Debug(DEBUG_DEVCON,("UniqueId Change error
37860|           | %08x\n",Irp->IoStatus.Status));
37861|       }

```

```

37849| } __except(
      | ExceptionFilter(GetExceptionInformation()) ) {
37850|     Debug(DEBUG_THREAD,("Uniqueld: Error! Exception
      | %08x\n", GetExceptionCode()));
37851| }
37852|
37853| if ( Irp->PendingReturned ) {
37854|     IoMarkIrpPending(Irp);
37855| }
37856|
37857| return STATUS_SUCCESS;
37858| }
37859|
37860| /*-----
      | -----*/
37861| STATIC NTSTATUS
37862| PSMANDiskQuerySuggestedLinkName(
37863|     IN PDEVICE_OBJECT
      | DeviceObject,
37864|     IN PIRP      Irp,
37865|     IN PVOID
      | /*Context*/
37866| )
37867|
37868| /*++
37869|
37870| Routine Description:
37871|
37872| This is the completion routine
37873|
37874| Called at <=DISPATCH_LEVEL
37875| Arguments:
37876|
37877| DeviceObject - Pointer to device object
37878| Irp          - IRP involved.
37879| Context      - NULL
37880|
37881| Return Value:
37882|
37883| NTSTATUS
37884|
37885| --*/
37886|
37887| {
37888|     PFILTERED_EXTENSION DevExt =
      | GetFilteredExtension(DeviceObject);
37889|
37890|     if ( Irp->IoStatus.Status==0 ) {
37891|         PMOUNTDEV_SUGGESTED_LINK_NAME Link=
      | (PMOUNTDEV_SUGGESTED_LINK_NAME)Irp->AssociatedIrp.System

```

```

    | Buffer;
37892|     Debug(DEBUG_DEVCON,("SuggestedLinkName: %d, %d:
    | '%-*.*ws"\n",Link->UseOnlyIfThereAreNoOtherLinks,Link->N
    | ameLength,Link->NameLength/2,Link->NameLength/2,Link->Na
    | me));
37893| }
37894|
37895|     if ( Irp->PendingReturned ) {
37896|         IoMarkIrpPending(Irp);
37897|     }
37898|
37899|     return STATUS_SUCCESS;;
37900| }
37901|
37902| /*-----
    | -----*/
37903| STATIC NTSTATUS
37904| PSMANDiskQueryDeviceName(
37905|     IN PDEVICE_OBJECT DeviceObject,
37906|     IN PIRP           Irp,
37907|     IN PVOID          /*Context*/
37908| )
37909|
37910| /*++
37911|
37912| Routine Description:
37913|
37914|     This is the completion routine
37915|
37916|     Called at <=DISPATCH_LEVEL
37917| Arguments:
37918|
37919|     DeviceObject - Pointer to device object
37920|     Irp          - IRP involved.
37921|     Context      - NULL
37922|
37923| Return Value:
37924|
37925|     NTSTATUS
37926|
37927| --*/
37928|
37929| {
37930|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DeviceObject);
37931|
37932|     if ( Irp->IoStatus.Status==0 ) {
37933|         PMOUNTDEV_NAME Name=
    | (PMOUNTDEV_NAME)Irp->AssociatedIrp.SystemBuffer;
37934|         Debug(DEBUG_DEVCON,("DeviceName %08x:

```

```

    | %-*.*ws\n",DeviceObject,Name->NameLength/2,Name->NameLen
    | gth/2,Name->Name));
37935|      // update what we know about the name incase it
    | changed
37936|
    | wcsncpy(DevExt->Name,Name->Name,Name->NameLength/sizeof(
    | WCHAR));
37937|
    | DevExt->Name[Name->NameLength/sizeof(WCHAR)]=L'\0';
37938|  }
37939|
37940|  if ( Irp->PendingReturned ) {
37941|      IoMarkIrpPending(Irp);
37942|  }
37943|
37944|  return STATUS_SUCCESS;
37945| }
37946| #endif
37947|
37948| /*-----
    | -----*/
37949| STATIC NTSTATUS
37950| PSMANDiskReserve(
37951|     IN PDEVICE_OBJECT DeviceObject,
37952|     IN PIRP          Irp,
37953|     IN PVOID          Context
37954| )
37955|
37956| {
37957|     Debug(DEBUG_DEVCON,("IOCTL_DISK_RESERVE %08x status
    | = %08x\n",DeviceObject,Irp->IoStatus.Status));
37958|
37959|     if ( Irp->PendingReturned ) {
37960|         IoMarkIrpPending(Irp);
37961|     }
37962|
37963|     return STATUS_SUCCESS;
37964| }
37965|
37966| /*-----
    | -----*/
37967| STATIC NTSTATUS
37968| PSMANDiskRelease(
37969|     IN PDEVICE_OBJECT DeviceObject,
37970|     IN PIRP          Irp,
37971|     IN PVOID          Context
37972| )
37973|
37974| {
37975|     Debug(DEBUG_DEVCON,("IOCTL_DISK_RELEASE %08x status

```

```

    | = %08x\n", DeviceObject, Irp->IoStatus.Status));
37976|
37977|     if ( Irp->PendingReturned ) {
37978|         IoMarkIrpPending(Irp);
37979|     }
37980|
37981|     return STATUS_SUCCESS;
37982| }
37983|
37984|
37985| /*-----
    | -----*/
37986| STATIC NTSTATUS
37987| PSMANDiskGetPartitionInfo(
37988|     IN PDEVICE_OBJECT
    | DeviceObject,
37989|     IN PIRP        Irp,
37990|     IN PVOID        /*Context*/
37991| )
37992|
37993| /*++
37994|
37995| Routine Description:
37996|
37997|     This is the completion routine for
    | IOCTL_DISK_GET_GEOMETRY.
37998|
37999|     Called at <=DISPATCH_LEVEL
38000| Arguments:
38001|
38002|     DeviceObject - Pointer to device object to being
    | shutdown by system.
38003|     Irp          - IRP involved.
38004|     Context      - NULL
38005|
38006| Return Value:
38007|
38008|     NTSTATUS
38009|
38010| --*/
38011|
38012| {
38013|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DeviceObject);
38014|
38015|     // if removable, adjust what we know about the
    | device
38016|     if ( Irp->IoStatus.Status==0 ) {
38017|         PPARTITION_INFORMATION partitionInfo=
    | (PPARTITION_INFORMATION)Irp->AssociatedIrp.SystemBuffer;

```

```

38018|
38019|     if ( partitionInfo ) {
38020|         Debug(DEBUG_DEVCON,("'%S': Offset=%08x%08x,
    | Length=%08x%08x, Hidden=%08x, Type=%02x\n",
38021|             DevExt->Name,
38022|
    | partitionInfo->StartingOffset.HighPart,
38023|
    | partitionInfo->StartingOffset.LowPart,
38024|
    | partitionInfo->PartitionLength.HighPart,
38025|
    | partitionInfo->PartitionLength.LowPart,
38026|
    | partitionInfo->HiddenSectors,
38027|
    | partitionInfo->PartitionType
38028|             ));
38029|
38030|         Debug(DEBUG_DEVCON,("
    | %08x%08x, Length=%08x%08x, Hidden=%08x, Type=%02x\n",
38031|
    | DevExt->Pi.StartingOffset.HighPart,
38032|
    | DevExt->Pi.StartingOffset.LowPart,
38033|
    | DevExt->Pi.PartitionLength.HighPart,
38034|
    | DevExt->Pi.PartitionLength.LowPart,
38035|
    | DevExt->Pi.HiddenSectors,
38036|
    | DevExt->Pi.PartitionType
38037|             ));
38038|
38039|         Debug(DEBUG_DEVCON,("  BI=%08x, RP=%08x,
    | RW=%08x, PN=%08x\n",
38040|
    | partitionInfo->BootIndicator,
38041|
    | partitionInfo->RecognizedPartition,
38042|
    | partitionInfo->RewritePartition,
38043|
    | partitionInfo->PartitionNumber
38044|             ));
38045|
38046|
38047|         // keep in sync with GET_PARTITION_INFO_EX
38048|         if(DevExt->Pi.PartitionLength.QuadPart <

```

```

    | partitionInfo->PartitionLength.QuadPart) {
38049|         LARGE_INTEGER New;
38050|
38051|         New.QuadPart =
    | partitionInfo->PartitionLength.QuadPart;
38052|         // need it in the number of sectors
38053|         New.QuadPart /= 512;
38054|
38055|         // Extend the bitmap
38056|         NTSTATUS Status =
    | ExtendFreeSpaceBitmaps( DeviceObject, New );
38057|         if(NT_SUCCESS(Status)) {
38058|             Debug(DEBUG_DEVCON,("Success
    | extending bitmaps\n"));
38059|         } else {
38060|             Debug(DEBUG_DEVCON,("Error %08x
    | extending bitmaps\n",Status));
38061|         }
38062|     }
38063|
38064|     // reupdate ourselves as it may have
    | changed.... (ftdisk, removable)
38065|     DevExt->Pi = *partitionInfo;
38066| } else {
38067|     Debug(DEBUG_DEVCON,("Error! buffer
    | NULL\n"));
38068| }
38069| }
38070|
38071| #if 0
38072|     if ( Irp->PendingReturned ) {
38073|         // Debug(DEBUG_DEVCON,("Marking Irp as
    | pending\n"));
38074|         IoMarkIrpPending(Irp);
38075|     }
38076| #endif
38077|
38078| // Debug(DEBUG_PROCCALL,("PSManDiskGetDriveLayout
    | Done\n"));
38079| return STATUS_SUCCESS;
38080| }
38081|
38082| /*-----
    | -----*/
38083| STATIC NTSTATUS
38084| PSManDiskGetPartitionInfoEx(
38085|     IN PDEVICE_OBJECT
    | DeviceObject,
38086|     IN PIRP      Irp,
38087|     IN PVOID     /*Context*/

```



```

38088|         )
38089|
38090| /*++
38091|
38092| Routine Description:
38093|
38094|     This is the completion routine for
38095|     | IOCTL_DISK_GET_GEOMETRY.
38096|     Called at <=DISPATCH_LEVEL
38097| Arguments:
38098|
38099|     DeviceObject - Pointer to device object to being
38100|     | shutdown by system.
38101|     Irp         - IRP involved.
38102|     Context     - NULL
38103| Return Value:
38104|
38105|     NTSTATUS
38106|
38107| --*/
38108|
38109| {
38110|     PFILTERED_EXTENSION DevExt =
38111|     | GetFilteredExtension(DeviceObject);
38112|     if(Irp->IoStatus.Status==0) {
38113|         PPARTITION_INFORMATION_EX partitionInfo=
38114|         | (PPARTITION_INFORMATION_EX)Irp->AssociatedIrp.SystemBuff
38115|         | er;
38116|         if (partitionInfo) {
38117|             #ifdef DEBUG
38118|             Debug(DEBUG_DEVCON,("'%S': Ps=%08x,
38119|             | So=%l64x, L=%l64x, pn=%d, rp=%d\n",
38120|             | DevExt->Name,
38121|             | partitionInfo->PartitionStyle,
38122|             | partitionInfo->StartingOffset,
38123|             | partitionInfo->PartitionLength,
38124|             | partitionInfo->PartitionNumber,
38125|             | partitionInfo->RewritePartition
38126|             | ));
38127|             if(partitionInfo->PartitionStyle==PARTITION_STYLE_MBR)
38128|             | {
38129|             Debug(DEBUG_DEVCON,(" Pt=%02x, bi=%d,
38130|             | rp=%d, hidden=%08x\n",
38131|             | partitionInfo->Mbr.PartitionType,
38132|             | partitionInfo->Mbr.BootIndicator,

```

```

38129|
| partitionInfo->Mbr.RecognizedPartition,
38130|         partitionInfo->Mbr.HiddenSectors
38131|     ));
38132|     } else
38133|
| if(partitionInfo->PartitionStyle==PARTITION_STYLE_GPT)
| {
38134|         UNICODE_STRING PT;
38135|         UNICODE_STRING ID;
38136|
| RtlStringFromGUID(partitionInfo->Gpt.PartitionType,&PT);
38137|
| RtlStringFromGUID(partitionInfo->Gpt.PartitionId,&ID);
38138|
38139|         Debug(DEBUG_DEVCON,(" Pt=%wZ, Id=%wZ,
| A=%!64x, Name='%S'\n",
38140|             &PT,
38141|             &ID,
38142|             partitionInfo->Gpt.Attributes,
38143|             partitionInfo->Gpt.Name
38144|         ));
38145|         RtlFreeUnicodeString(&PT);
38146|         RtlFreeUnicodeString(&ID);
38147|     } else {
38148|         ASSERT(FALSE);
38149|         Debug(DEBUG_DEVCON,("Unknown partition
| type\n"));
38150|     }
38151| #endif
38152|         // reupdate ourselves as it may have
| changed.... (ftdisk, removable)
38153|
38154|         DevExt->Pi.StartingOffset =
| partitionInfo->StartingOffset;
38155|
38156|         // keep in sync with GET_PARTITION_INFO
38157|         if(DevExt->Pi.PartitionLength.QuadPart <
| partitionInfo->PartitionLength.QuadPart) {
38158|             LARGE_INTEGER New;
38159|
38160|             New.QuadPart =
| partitionInfo->PartitionLength.QuadPart;
38161|             // need it in the number of sectors
38162|             New.QuadPart /= 512;
38163|
38164|             // Extend the bitmap
38165|             NTSTATUS Status =
| ExtendFreeSpaceBitmaps( DeviceObject, New );
38166|             if(NT_SUCCESS(Status)) {

```

```

38167|         Debug(DEBUG_DEVCON,("Success
| extending bitmaps\n"));
38168|     } else {
38169|         Debug(DEBUG_DEVCON,("Error %08x
| extending bitmaps\n",Status));
38170|     }
38171| }
38172|
38173|     DevExt->Pi.PartitionLength =
| partitionInfo->PartitionLength;
38174|
| DevExt->Pi.PartitionNumber=partitionInfo->PartitionNumbe
| r;
38175|
| DevExt->Pi.RewritePartition=partitionInfo->RewritePartit
| ion;
38176|
38177|
| if(partitionInfo->PartitionStyle==PARTITION_STYLE_MBR)
| {
38178|         // Specific to MBR partition onlys
38179|
| DevExt->Pi.HiddenSectors=partitionInfo->Mbr.HiddenSector
| s;
38180|
| DevExt->Pi.PartitionType=partitionInfo->Mbr.PartitionTyp
| e;
38181|
| DevExt->Pi.BootIndicator=partitionInfo->Mbr.BootIndicato
| r;
38182|
| DevExt->Pi.RecognizedPartition=partitionInfo->Mbr.Recogn
| izedPartition;
38183|     } else
38184|
| if(partitionInfo->PartitionStyle==PARTITION_STYLE_GPT)
| {
38185|         // we dont support this now
38186|         ASSERT(FALSE);
38187|         DevExt->Pi.HiddenSectors=0;
38188|         DevExt->Pi.PartitionType=0;
38189|         DevExt->Pi.BootIndicator=0;
38190|         DevExt->Pi.RecognizedPartition=0;
38191|     } else {
38192|         ASSERT(FALSE);
38193|     }
38194| } else {
38195|     Debug(DEBUG_DEVCON,("Error! buffer
| NULL\n"));
38196| }

```

```

38197|    }
38198|
38199| #if 0
38200|     if (Irp->PendingReturned) {
38201|         //      Debug(DEBUG_DEVCON,("Marking Irp as
           | pending\n"));
38202|         IoMarkIrpPending(Irp);
38203|     }
38204| #endif
38205|
38206| //      Debug(DEBUG_PROCCALL,("OtManDiskGetDriveLayout
           | Done\n"));
38207|     return STATUS_SUCCESS;
38208| }
38209|
38210|
38211| /*-----
           | -----*/
38212| STATIC NTSTATUS
38213| PSManDeviceControlDevice(
38214|         PDEVICE_OBJECT DeviceObject,
38215|         PIRP Irp
38216|         )
38217|
38218| /*++
38219|
38220| Routine Description:
38221|
38222|     This device control dispatcher handles only the
           | disk performance
38223|     device control. All others are passed down to the
           | disk drivers.
38224|     The disk performane device control returns a
           | current snapshot of
38225|     the performance data.
38226|
38227| Arguments:
38228|
38229|     DeviceObject - Context for the activity.
38230|     Irp          - The device control argument block.
38231|
38232| Return Value:
38233|
38234|     Status is returned.
38235|
38236| --*/
38237|
38238| {
38239|     PFILTERED_EXTENSION DevExt =
           | GetFilteredExtension(DeviceObject);

```

```

38240| // KIRQL          currentIrql;
38241| // PIO_STACK_LOCATION nextIrpsStack =
    | IoGetNextIrpsStackLocation(Irp);
38242| PIO_STACK_LOCATION currentIrpsStack =
    | IoGetCurrentIrpsStackLocation(Irp);
38243| NTSTATUS Status;
38244|
38245| TRACE( TRACE_IOCTL,
38246|
    | currentIrpsStack->Parameters.Read.ByteOffset.HighPart,
38247|
    | currentIrpsStack->Parameters.Read.ByteOffset.LowPart,
38248|     currentIrpsStack->Parameters.Read.Length,
38249|     currentIrpsStack->Parameters.Read.Key,
38250|     "");
38251|
38252| #ifdef DEBUG
38253|     if ( 1 ) {
38254|
38255|         | switch(currentIrpsStack->Parameters.DeviceIoControl.IoCon
    | trolCode) {
38256|             case IOCTL_STORAGE_RESERVE :
38257|             case IOCTL_DISK_RESERVE :
38258|             case IOCTL_STORAGE_RELEASE :
38259|             case IOCTL_DISK_RELEASE :
38260|             case IOCTL_DISK_CHECK_VERIFY :
38261|                 // dont display stuff on these ioctls as
    | there are a lot of them
38262|                 break;
38263|             default:
38264|                 Debug(DEBUG_DEVCON |
    | DEBUG_PROCCALL,("PManDeviceControlDevice Device = %p,
    | Irp = %p, Flags=%08x\n",
38265|
    | DeviceObject, Irp, Irp->Flags));
38266|
38267|                 Debug(DEBUG_DEVCON | DEBUG_PROCCALL,(" %s
    | Major=%d, Minor=%d, Flgs=%08x, Ctrl=%08x\n",
38268|
    | File_GetMajorFunctionName(currentIrpsStack->MajorFunction
    | ),
38269|
    | currentIrpsStack->MajorFunction,
38270|
    | currentIrpsStack->MinorFunction,
38271|
    | currentIrpsStack->Flags,
38272|
    | currentIrpsStack->Control

```

```

38273|         ));
38274|
38275|         Debug(DEBUG_DEVCON,(" OutputLen=%d,
| InputLen=%d, Control=%08lx (%s), Type3=%p\n",
38276|         | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
| Length,
38277|         | currentIrpStack->Parameters.DeviceIoControl.InputBufferL
| ength,
38278|         | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
| e,
38279|         | File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
| Control.IoControlCode),
38280|         | currentIrpStack->Parameters.DeviceIoControl.Type3InputBu
| ffer)
38281|         );
38282|         break;
38283|     }
38284|
38285| }
38286| #endif
38287|
38288| // say an io is happening. we need this for ioctls
| because the disk partitions may change.
38289| GetGlobalDeviceForRead();
38290|
38291| // catch certain ioctls so we can do pre/post
| processing.
38292| switch (
| currentIrpStack->Parameters.DeviceIoControl.IoControlCod
| e ) {
38293|     case IOCTL_DISK_SET_PARTITION_INFO_EX: {
38294|         PSET_PARTITION_INFORMATION_EX PartInfo
| =
| (PSET_PARTITION_INFORMATION_EX)Irp->AssociatedIrp.System
| Buffer;
38295|
| if(PartInfo->PartitionStyle==PARTITION_STYLE_MBR) {
38296|         Debug(DEBUG_DEVCON,("Devcon:
| Partition changed to %02x from
| %02x\n",PartInfo->Mbr.PartitionType,DevExt->Pi.Partition
| Type));
38297|         DevExt->Pi.PartitionType =
| PartInfo->Mbr.PartitionType;
38298|     } else
38299|

```

```

    | if(PartInfo->PartitionStyle==PARTITION_STYLE_GPT) {
38300|         // we dont support this right now
38301|         ASSERT(FALSE);
38302|     } else {
38303|         ASSERT(FALSE);
38304|     }
38305|     // continue with passthru
38306|     break;
38307| }
38308| // changes partition type. usually after a
    | format.
38309| case IOCTL_DISK_SET_PARTITION_INFO: {
38310|     PSET_PARTITION_INFORMATION PartInfo =
    | (PSET_PARTITION_INFORMATION)Irp->AssociatedIrp.SystemBuf
    | fer;
38311|     Debug(DEBUG_DEVCON,("Devcon: Partition
    | changed to %02x from
    | %02x\n",PartInfo->PartitionType,DevExt->Pi.PartitionType
    | ));
38312|     DevExt->Pi.PartitionType =
    | PartInfo->PartitionType;
38313|     // continue with passthru
38314|     break;
38315| }
38316| // a new partition has been added or an
    | existing one deleted.
38317| case IOCTL_DISK_SET_DRIVE_LAYOUT: {
38318|     CCHAR
    | boost=IO_NO_INCREMENT;
38319|
38320|     Irp->IoStatus.Status = Status =
    | PSMAN_FORWARD_IRP_SYNCHRONOUS(DeviceObject,Irp);
38321|
38322|     if ( NT_SUCCESS(Status) ) {
38323|
38324|         //
38325|         // Process the new partition table.
    | The work for the
38326|         // set drive layout was done
    | synchronously because this
38327|         // routine performs synchronous
    | activities.
38328|         //
38329|         Debug(DEBUG_DEVCON,("Devcon:
    | Success setting drive layout\n"));
38330|
38331| #if _WIN32_WINNT < 0x0500
38332|         // Add/remove called for pnp
    | devices
38333|

```

```

    | PSMANMakePartitionObjects(DevExt->PhysicalDevice,FALSE);
38334| #endif
38335|         boost = IO_DISK_INCREMENT;
38336|     } else {
38337|         boost = IO_NO_INCREMENT;
38338|         Debug(DEBUG_DEVCON,("Devcon: Error
    | %08x setting drive layout\n",Status));
38339|     }
38340|
38341|     ReleaseGlobalDeviceForRead();
38342|     IoCompleteRequest(Irp, boost);
38343|     return Status;
38344|
38345| }
38346|
38347| case IOCTL_STORAGE_FIND_NEW_DEVICES:
38348| case IOCTL_DISK_FIND_NEW_DEVICES: {
38349|     CCHAR
    | boost=IO_NO_INCREMENT;
38350|     ULONG      Save =
    | IoGetConfigurationInformation()->DiskCount;
38351|
38352|     Debug(DEBUG_DEVCON,("DevCon:
    | Find_new_devices: Old disk count=%d\n",Save));
38353|     Irp->IoStatus.Status = Status =
    | PSMANForwardIrpSynchronous(DeviceObject,Irp);
38354|
38355|     if ( NT_SUCCESS(Status) ) {
38356|         Debug(DEBUG_DEVCON,("DevCon:
    | Find_new_devices: New disk
    | count=%d\n",IoGetConfigurationInformation()->DiskCount))
    | ;
38357| #if _WIN32_WINNT < 0x0500
38358|         // what to do for win2k?? FIXFIXFIX
38359|
    | PSMANInitialize(DeviceObject->DriverObject,
    | (PVOID)Save, 0);
38360| #endif
38361|         boost = IO_DISK_INCREMENT;
38362|     }
38363|
38364|     // Call target driver.
38365|     ReleaseGlobalDeviceForRead();
38366|     IoCompleteRequest(Irp, boost);
38367|     return Status;
38368| }
38369| #ifdef DEBUG
38370| case IOCTL_SCSI_PASS_THROUGH_DIRECT: {
38371|     // dump what command is being sent to
    | lower driver.

```



```

38372|         PSCSI_PASS_THROUGH_DIRECT sptd =
| (PSCSI_PASS_THROUGH_DIRECT)Irp->AssociatedIrp.SystemBuff
| er;
38373|         Debug(DEBUG_DEVCON,("SPTD: %d: Cdb =
| %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
| %02x\n",sptd->CdbLength,
38374|
| sptd->Cdb[0],sptd->Cdb[1],sptd->Cdb[2],sptd->Cdb[3],sptd
| ->Cdb[4],sptd->Cdb[5],
38375|
| sptd->Cdb[6],sptd->Cdb[7],sptd->Cdb[8],sptd->Cdb[9],sptd
| ->Cdb[10],sptd->Cdb[11]
38376|         ));
38377|
38378|         break;
38379|     }
38380|     case IOCTL_SCSI_PASS_THROUGH: {
38381|         // dump what command is being sent to
| lower driver.
38382|         PSCSI_PASS_THROUGH spt =
| (PSCSI_PASS_THROUGH)Irp->AssociatedIrp.SystemBuffer;
38383|         Debug(DEBUG_DEVCON,("SPT: %d: Cdb =
| %02x %02x %02x %02x %02x %02x %02x %02x %02x %02x
| %02x\n",spt->CdbLength,
38384|
| spt->Cdb[0],spt->Cdb[1],spt->Cdb[2],spt->Cdb[3],spt->Cdb
| [4],spt->Cdb[5],
38385|
| spt->Cdb[6],spt->Cdb[7],spt->Cdb[8],spt->Cdb[9],spt->Cdb
| [10],spt->Cdb[11]
38386|         ));
38387|
38388|         break;
38389|     }
38390|     case IOCTL_DISK_GET_DRIVE_LAYOUT: {
38391|         // Copy current stack to next stack.
38392|
| IoCopyCurrentIrpStackLocationToNext(Irp);
38393|
38394|         // Ask to be called back during request
| completion.
38395|         IoSetCompletionRoutine(Irp,
38396|
| PSMANDiskGetDriveLayout,
38397|
| NULL,
38398|
| TRUE,
38399|
| TRUE,
38400|
| TRUE);
38401|
38402|         // Call target driver.

```

```

38403|         ReleaseGlobalDeviceForRead();
38404|         return
    | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38405|     }
38406| #endif
38407|     case CTL_CODE(IOCTL_DISK_BASE, 0x0012,
    | METHOD_BUFFERED, FILE_READ_ACCESS):
38408|     case IOCTL_DISK_GET_PARTITION_INFO_EX : {
38409| #if 0
38410|         // Copy current stack to next stack.
38411|         IoCopyCurrentIrpStackLocationToNext(Irp);
38412|
38413|         // Ask to be called back during request
    | completion.
38414|         IoSetCompletionRoutine(Irp,
38415|
    | PSMANDiskGetPartitionInfoEx,
38416|             NULL,
38417|             TRUE,
38418|             TRUE,
38419|             TRUE);
38420|
38421|         // Call target driver.
38422|         ReleaseGlobalDeviceForRead();
38423|         return
    | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38424| #else
38425|         Irp->IoStatus.Status = Status =
    | PSMANForwardIrpSynchronous(DeviceObject,Irp);
38426|
38427|         if ( NT_SUCCESS(Status) ) {
38428|             Debug(DEBUG_DEVCON,("Devcon:
    | Success getting part info ex\n"));
38429|             Status =
    | PSMANDiskGetPartitionInfoEx(DeviceObject,Irp,NULL);
38430|             if ( NT_SUCCESS(Status) ) {
38431|                 Debug(DEBUG_DEVCON,("Devcon:
    | Success processing part info ex\n"));
38432|             } else {
38433|                 Debug(DEBUG_DEVCON,("Devcon:
    | Error %08x processing part info ex\n",Status));
38434|             }
38435|         } else {
38436|             Debug(DEBUG_DEVCON,("Devcon: Error
    | %08x getting part info\n",Status));
38437|         }
38438|
38439|         ReleaseGlobalDeviceForRead();
38440|         IoCompleteRequest(Irp,
    | IO_DISK_INCREMENT);

```

```

38441|         return Status;
38442| #endif
38443|     }
38444|     case IOCTL_DISK_GET_PARTITION_INFO: {
38445| #if 0
38446|         // Copy current stack to next stack.
38447|
38448|         | IoCopyCurrentIrpStackLocationToNext(Irp);
38449|         // Ask to be called back during request
38450|         | completion.
38451|         IoSetCompletionRoutine(Irp,
38452|         | PManDiskGetPartitionInfo,
38453|         NULL,
38454|         TRUE,
38455|         TRUE,
38456|         TRUE);
38457|         // Call target driver.
38458|         ReleaseGlobalDeviceForRead();
38459|         return
38460|         | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38461| #else
38462|         Irp->IoStatus.Status = Status =
38463|         | PManForwardIrpSynchronous(DeviceObject,Irp);
38464|         if ( NT_SUCCESS(Status) ) {
38465|             Debug(DEBUG_DEVCON,("Devcon:
38466|             | Success getting part info\n"));
38467|             Status =
38468|             | PManDiskGetPartitionInfo(DeviceObject,Irp,NULL);
38469|             if ( NT_SUCCESS(Status) ) {
38470|                 Debug(DEBUG_DEVCON,("Devcon:
38471|                 | Success processing part info\n"));
38472|             } else {
38473|                 Debug(DEBUG_DEVCON,("Devcon:
38474|                 | Error %08x processing part info\n",Status));
38475|             }
38476|         } else {
38477|             Debug(DEBUG_DEVCON,("Devcon: Error
38478|             | %08x getting part info\n",Status));
38479|         }
38480|         ReleaseGlobalDeviceForRead();
38481|         IoCompleteRequest(Irp,
38482|         | IO_DISK_INCREMENT);
38483|         return Status;
38484|     }
38485| #endif

```

```

38480|
38481|     }
38482| #if _WIN32_WINNT >= 0x0500
38483|     case IOCTL_MOUNTDEV_QUERY_DEVICE_NAME: {
38484|         // Copy current stack to next stack.
38485|
38486|         | IoCopyCurrentIrpStackLocationToNext(Irp);
38487|         // Ask to be called back during request
38488|         | completion.
38489|         IoSetCompletionRoutine(Irp,
38490|                                NULL,
38491|                                TRUE,
38492|                                TRUE,
38493|                                TRUE);
38494|
38495|         // Call target driver.
38496|         ReleaseGlobalDeviceForRead();
38497|         return
38498|         | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38499|     }
38500|     case IOCTL_MOUNTDEV_QUERY_UNIQUE_ID: {
38501|         // Copy current stack to next stack.
38502|
38503|         | IoCopyCurrentIrpStackLocationToNext(Irp);
38504|         // Ask to be called back during request
38505|         | completion.
38506|         IoSetCompletionRoutine(Irp,
38507|                                NULL,
38508|                                TRUE,
38509|                                TRUE,
38510|                                TRUE);
38511|
38512|         // Call target driver.
38513|         ReleaseGlobalDeviceForRead();
38514|         return
38515|         | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38516|     }
38517|     case IOCTL_MOUNTDEV_UNIQUE_ID_CHANGE_NOTIFY: {
38518|         // Copy current stack to next stack.
38519|
38520|         | IoCopyCurrentIrpStackLocationToNext(Irp);
38521|         // Ask to be called back during request
38522|         | completion.

```

```

38520|         IoSetCompletionRoutine(Irp,
38521|         | PSMANDiskUniqueIdChangeNotify,
38522|         NULL,
38523|         TRUE,
38524|         TRUE,
38525|         TRUE);
38526|
38527|         // Call target driver.
38528|         ReleaseGlobalDeviceForRead();
38529|         return
38530|         | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38531|     }
38532|     case IOCTL_MOUNTDEV_QUERY_SUGGESTED_LINK_NAME:
38533|     | {
38534|         // Copy current stack to next stack.
38535|         | IoCopyCurrentIrpStackLocationToNext(Irp);
38536|         // Ask to be called back during request
38537|         | completion.
38538|         IoSetCompletionRoutine(Irp,
38539|         | PSMANDiskQuerySuggestedLinkName,
38540|         NULL,
38541|         TRUE,
38542|         TRUE,
38543|         TRUE);
38544|         // Call target driver.
38545|         ReleaseGlobalDeviceForRead();
38546|         return
38547|         | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38548|     }
38549|     case IOCTL_MOUNTDEV_LINK_CREATED: {
38550|         PMOUNTDEV_NAME Link=
38551|         | (PMOUNTDEV_NAME)Irp->AssociatedIrp.SystemBuffer;
38552|         Debug(DEBUG_DEVCON,("Link created:
38553|         | %08x:
38554|         | '%-*.*ws'\n", DeviceObject, Link->NameLength/2, Link->NameL
38555|         | ength/2, Link->Name));
38556|
38557| #define MY_MOUNTMGR_IS_VOLUME_NAME(s) (
38558|     | \
38559|     ((s)->NameLength == 96 ||
38560|     | ((s)->NameLength == 98 && (s)->Name[48] == '\\')) && \
38561|     (s)->Name[0] == '\\' &&
38562|     | \
38563|     ((s)->Name[1] == '?' || (s)->Name[1]
38564|     | == '\\') &&

```

```

38555|         (s)->Name[2] == '?' &&
      | \
38556|         (s)->Name[3] == '\\' &&
      | \
38557|         (s)->Name[4] == 'V' &&
      | \
38558|         (s)->Name[5] == 'o' &&
      | \
38559|         (s)->Name[6] == 'l' &&
      | \
38560|         (s)->Name[7] == 'u' &&
      | \
38561|         (s)->Name[8] == 'm' &&
      | \
38562|         (s)->Name[9] == 'e' &&
      | \
38563|         (s)->Name[10] == '{' &&
      | \
38564|         (s)->Name[19] == '-' &&
      | \
38565|         (s)->Name[24] == '-' &&
      | \
38566|         (s)->Name[29] == '-' &&
      | \
38567|         (s)->Name[34] == '-' &&
      | \
38568|         (s)->Name[47] == '}'
      | \
38569|     )
38570|
38571|         // 96:
      | \"?Volume{c8e56d0c-93c5-11d4-9910-806d6172696f}'
38572|         // 28: '\DosDevices\C:'
38573|         // 96:
      | \"?Volume{c8e56d0d-93c5-11d4-9910-806d6172696f}'
38574|         //
      | 0123456789-123456789-123456789-123456789-12345678
38575|         //
      | 0123456789-123456789-123456789-1234567
38576|
38577|
38578|         if(DevExt->VolumeId==0) {
38579|             if (
      | MY_MOUNTMGR_IS_VOLUME_NAME(Link) ) {
38580|                 WCHAR *p=Link->Name+11;
38581|                 RtlCopyMemory (
      | DevExt->VolumeGuid, p, 36*sizeof(WCHAR) );
38582|                 DevExt->VolumeGuid[36] = 0;
38583|
38584|                 // is a mount manager point

```

```

38585|             DevExt->Volumeld =
| WAsciiToInt(&p,16);
38586|             Debug(DEBUG_DEVCON,("Mount
| manager, storing volume guid
| %08x\n",DevExt->Volumeld));
38587|         }
38588|     } else {
38589|         Debug(DEBUG_DEVCON,("Mount manager,
| already have volume guid '%S'\n",DevExt->VolumeGuid));
38590|     }
38591|     break;
38592| }
38593| case IOCTL_MOUNTDEV_LINK_DELETED: {
38594|     PMOUNTDEV_NAME Link=
| (PMOUNTDEV_NAME)Irp->AssociatedIrp.SystemBuffer;
38595|     Debug(DEBUG_DEVCON,("Link deleted:
| %08x:
| '%-*.*ws'\n",DeviceObject,Link->NameLength/2,Link->NameL
| ength/2,Link->Name));
38596|     if ( MY_MOUNTMGR_IS_VOLUME_NAME(Link) )
| {
38597|         WCHAR *p=Link->Name+11;
38598|         ULONG Id = WAsciiToInt(&p,16);
38599|
38600|         // told to delete the one we are
| using..
38601|         // FIXFIXFIX what to do if they
| never set the link again
38602|         // maybe because we got several
| link creates before this delete
38603|         // occurred
38604|         if(Id==DevExt->Volumeld) {
38605|             DevExt->Volumeld = 0;
38606|
| wcsncpy(DevExt->VolumeGuid,L"_GUID_Deleted_");
38607|         }
38608|     }
38609|     break;
38610| }
38611| case IOCTL_VOLUME_ONLINE : {
38612|     Irp->IoStatus.Status = Status =
| PSMAN_FORWARD_IRP_SYNCHRONOUS(DeviceObject,Irp);
38613|     Debug(DEBUG_DEVCON,("DevCon:
| VolumeOnline status=%08x, phy=%d,
| directio=%d\n",Status,DevExt->IsPhysical,DevExt->DoDirec
| tIO));
38614| #if 0
38615|     if(DevExt->IsPhysical) {
38616|         | PersistentDictionary::RetrieveDirectIOMaps(DeviceObject)

```

```

| ;
38617|             if(DevExt->Cache.HeaderFile.Direct)
| {
38618|                 Debug(DEBUG_DEVCON,("DevCon:
| VolumeOnline directio=%d, setting to
| true\n",DevExt->DoDirectIO));
38619|                 DevExt->DoDirectIO = TRUE;
38620|                 Status =
| PersistentDictionary::LoadSnapShotsForVolume
| (DeviceObject,TRUE,NULL );
38621|                 Debug(DEBUG_DEVCON,("DevCon:
| load snapshots returned %08x\n",Status));
38622|                 Status=STATUS_SUCCESS;
38623|             }
38624|         }
38625| #endif
38626|         ReleaseGlobalDeviceForRead();
38627|         IoCompleteRequest(Irp,
| IO_NO_INCREMENT);
38628|         return Status;
38629|     }
38630|     case IOCTL_VOLUME_OFFLINE : {
38631|         Irp->IoStatus.Status = Status =
| PSMANForwardIrpSynchronous(DeviceObject,Irp);
38632|
38633|         Debug(DEBUG_DEVCON,("DevCon:
| VolumeOffline status=%08x\n",Status));
38634|         ReleaseGlobalDeviceForRead();
38635|         IoCompleteRequest(Irp,
| IO_NO_INCREMENT);
38636|         return Status;
38637|     }
38638|     case IOCTL_STORAGE_RESERVE :
38639|     case IOCTL_DISK_RESERVE : {
38640|         // Copy current stack to next stack.
38641|
| IoCopyCurrentIrpStackLocationToNext(Irp);
38642|
38643|         // Ask to be called back during request
| completion.
38644|         IoSetCompletionRoutine(Irp,
38645|         | PSMANDiskReserve,
38646|         NULL,
38647|         TRUE,
38648|         TRUE,
38649|         TRUE);
38650|
38651|         // Call target driver.
38652|         ReleaseGlobalDeviceForRead();

```



```

38653|         return
    | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38654|     }
38655|     case IOCTL_STORAGE_RELEASE :
38656|     case IOCTL_DISK_RELEASE : {
38657|         // Copy current stack to next stack.
38658|
    | IoCopyCurrentIrpStackLocationToNext(Irp);
38659|
38660|         // Ask to be called back during request
    | completion.
38661|         IoSetCompletionRoutine(Irp,
38662|
    | PManDiskRelease,
38663|
        NULL,
38664|
        TRUE,
38665|
        TRUE,
38666|
        TRUE);
38667|
38668|         // Call target driver.
38669|         ReleaseGlobalDeviceForRead();
38670|         return
    | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38671|     }
38672|
38673|     case IOCTL_STORAGE_GET_DEVICE_NUMBER: {
38674|         // Copy current stack to next stack.
38675|
    | IoCopyCurrentIrpStackLocationToNext(Irp);
38676|
38677|         // Ask to be called back during request
    | completion.
38678|         IoSetCompletionRoutine(Irp,
38679|
    | PManDiskGetDeviceNumber,
38680|
        NULL,
38681|
        TRUE,
38682|
        TRUE,
38683|
        TRUE);
38684|
38685|         // Call target driver.
38686|         ReleaseGlobalDeviceForRead();
38687|         return
    | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38688|     }
38689|     case IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS: {
38690|         // Copy current stack to next stack.
38691|
    | IoCopyCurrentIrpStackLocationToNext(Irp);
38692|

```

```

38693|          // Ask to be called back during request
      | completion.
38694|          IoSetCompletionRoutine(Irp,
38695|          | PManDiskGetDiskExtents,
38696|          NULL,
38697|          TRUE,
38698|          TRUE,
38699|          TRUE);
38700|
38701|          // Call target driver.
38702|          ReleaseGlobalDeviceForRead();
38703|          return
      | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38704|      }
38705|      case IOCTL_VOLUME_QUERY_VOLUME_NUMBER: {
38706|          // Copy current stack to next stack.
38707|
      | IoCopyCurrentIrpStackLocationToNext(Irp);
38708|
38709|          // Ask to be called back during request
      | completion.
38710|          IoSetCompletionRoutine(Irp,
38711|          | PManDiskGetVolumeNumber,
38712|          NULL,
38713|          TRUE,
38714|          TRUE,
38715|          TRUE);
38716|
38717|          // Call target driver.
38718|          ReleaseGlobalDeviceForRead();
38719|          return
      | IoCallDriver(DevExt->TargetDeviceObject, Irp);
38720|      }
38721| #endif
38722|      case IOCTL_STORAGE_GET_MEDIA_TYPES:
38723|      case IOCTL_DISK_GET_MEDIA_TYPES:
38724|      case IOCTL_DISK_GET_DRIVE_GEOMETRY: {
38725|          // Copy current stack to next stack.
38726|
      | IoCopyCurrentIrpStackLocationToNext(Irp);
38727|
38728|          // Ask to be called back during request
      | completion.
38729|          IoSetCompletionRoutine(Irp,
38730|          | PManDiskGetGeometry,
38731|          NULL,
38732|          TRUE,

```



```

38767|                // add/remove called for
| pnp devices
38768|
| PSMANMakePartitionObjects(DevExt->PhysicalDevice,FALSE);
38769| #endif
38770|                }
38771|                DevExt->ChangeCount=0;
38772|                }
38773|        } else {
38774|                if ( DevExt->PSMed ) {
38775|                Debug(DEBUG_DEVCON,("DevCon:
| DiskCheck: Psm is on but device has changed %08x
| %08x\n",Status,*ChangeCount));
38776|                // psm is open, stop it since
| this device is being psmmed, and
38777|                // it has just (possibly) been
| changed on us.
38778|                // FIXFIXFIX what to do when
| this happens?
38779|
| //FailRequest(NULL,STATUS_MEDIA_CHANGED);
38780|                }
38781|                // cant be ++ as we may stay in
| sync with the disk driver ;(
38782|                // we also cant make 0 or when
| there is no counter we wont check,
38783|                // 1 is what the disk manage starts
| out at, we start 0, so this should force a
38784|                // relook of the disk
38785|                DevExt->ChangeCount=1;
38786|                }
38787|                ReleaseGlobalDeviceForRead();
38788|                IoCompleteRequest(Irp,
| IO_NO_INCREMENT);
38789|                return Status;
38790|        }
38791|        default:
38792|                // nothing to do..
38793|                break;
38794|    }
38795|
38796|    // okay, its an ioctl we do not need (or care) to
| look at,
38797|    // so pass it on thru.
38798|
38799|    ReleaseGlobalDeviceForRead();
38800|    Status = PSMANPassThru( DeviceObject, Irp );
38801| #ifdef DEBUG
38802|    if ( 1 ) {
38803|        Debug(DEBUG_DEVCON |

```

```

    | DEBUG_PROCCALL,("PSManDeviceControlDevice Done
    | Device=%p, Status=%08x\n",DeviceObject,Status));
38804|    }
38805| #endif
38806|    return Status;
38807|
38808| } // end PSManDeviceControlDevice()
38809|
38810| /*-----
    | -----*/
38811| STATIC NTSTATUS PSManDeviceControlVDisk(
38812|     PDEVICE_OBJECT
    | DeviceObject,
38813|     PIRP Irp
38814| )
38815| {
38816|     PVDISK_EXTENSION DevExt =
    | GetVDiskExtension(DeviceObject);
38817|     PIO_STACK_LOCATION currentIrpStack =
    | IoGetCurrentIrpStackLocation( Irp );
38818|     NTSTATUS Status =
    | STATUS_INVALID_DEVICE_REQUEST;
38819|     CHAR IoIncrement=IO_NO_INCREMENT;
38820|     BOOLEAN CompleteRequest = TRUE;
38821|
38822|     #if DO_ALL_IO
38823|     if (
    | (currentIrpStack->Parameters.DeviceIoControl.IoControlCo
    | de!=IOCTL_DISK_CHECK_VERIFY) &&
38824|
    | (currentIrpStack->Parameters.DeviceIoControl.IoControlCo
    | de!=IOCTL_STORAGE_CHECK_VERIFY) ) {
38825|         Debug(DEBUG_DEVCON |
    | DEBUG_PROCCALL,("PSManDeviceControlVDisk: Dev=%p,
    | Ioctl=%08x - %s\n",
38826|
    | DeviceObject,
38827|
    | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
    | e,
38828|
    | File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
    | Control.IoControlCode)));
38829|     }
38830|     #endif /*DO_ALL_IO*/
38831|
38832|     Irp->IoStatus.Information = 0;
38833|     IoIncrement = IO_NO_INCREMENT;
38834|
38835|     AcquireVDiskResource();

```

```

38836|  __try {
38837|      __try {
38838|          switch (
            | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
            | e ) {
38839|              case IOCTL_STORAGE_EJECT_MEDIA:
38840|              case IOCTL_DISK_EJECT_MEDIA : {
38841|                  #if DO_ALL_IO
38842|
            | Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
            | Ioctl=%08x - %s\n",
38843|
            | DeviceObject,
38844|
            | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
            | e,
38845|
            | File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
            | Control.IoControlCode)));
38846|                #endif /*DO_ALL_IO*/
38847|                DevExt->DriveNotReady = TRUE;
38848|                DevExt->LockCount=0;
38849|                Irp->IoStatus.Status =
            | STATUS_SUCCESS;
38850|                break;
38851|            }
38852|            case IOCTL_STORAGE_LOAD_MEDIA :
38853|            case IOCTL_DISK_LOAD_MEDIA : {
38854|                #if DO_ALL_IO
38855|
            | Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
            | Ioctl=%08x - %s\n",
38856|
            | DeviceObject,
38857|
            | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
            | e,
38858|
            | File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
            | Control.IoControlCode)));
38859|                #endif /*DO_ALL_IO*/
38860|                DevExt->DriveNotReady = FALSE;
38861|                DevExt->LockCount=0;
38862|                Irp->IoStatus.Status =
            | STATUS_SUCCESS;
38863|                break;
38864|            }
38865|            case IOCTL_STORAGE_RESERVE :
38866|            case IOCTL_DISK_RESERVE : {
38867|                #if DO_ALL_IO

```

```

38868|
| Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
| Ioctl=%08x - %s\n",
38869|
| DeviceObject,
38870|
| currentIrpStack->Parameters.DeviceIoControl.IoControlCod
| e,
38871|
| File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
| Control.IoControlCode)));
38872|                #endif /*DO_ALL_IO*/
38873|                Irp->IoStatus.Status =
| STATUS_SUCCESS;
38874|                break;
38875|                }
38876|                case IOCTL_STORAGE_RELEASE :
38877|                case IOCTL_DISK_RELEASE : {
38878|                #if DO_ALL_IO
38879|
| Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%,
| Ioctl=%08x - %s\n",
38880|
| DeviceObject,
38881|
| currentIrpStack->Parameters.DeviceIoControl.IoControlCod
| e,
38882|
| File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
| Control.IoControlCode)));
38883|                #endif /*DO_ALL_IO*/
38884|                Irp->IoStatus.Status =
| STATUS_SUCCESS;
38885|                break;
38886|                }
38887|                // for removable media..
38888|                case IOCTL_STORAGE_CHECK_VERIFY :
38889|                case IOCTL_DISK_CHECK_VERIFY : {
38890|
38891|                // If a buffer for a media
| change count was provided, make sure it's
38892|                // big enough to hold the
| result
38893| #if 0
38894|
| Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
| Ioctl=%08x - %s\n",
38895|
| DeviceObject,
38896|

```

```

| currentIrpStack->Parameters.DeviceIoControl.IoControlCod
| e,
38897|
| File_GetIOCTLString(currentIrpStack->Parameters.DeviceIo
| Control.IoControlCode)));
38898| #endif
38899|
38900|         if (
| currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
| Length ) {
38901|             ULONG *outputBuffer=NULL;
38902|
38903|             // If the buffer is too
| small to hold the media change count
38904|             // then return an error to
| the caller
38905|
38906|             if (
| currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
| Length < sizeof(ULONG) ) {
38907|                 Irp->IoStatus.Status =
| STATUS_BUFFER_TOO_SMALL;
38908|                 break;
38909|             }
38910|
38911|             Irp->IoStatus.Status =
| CheckMediaLoaded(DeviceObject,Irp);
38912|             // The caller has provided
| a valid buffer.
38913|             outputBuffer =
| (ULONG*)Irp->AssociatedIrp.SystemBuffer;
38914|             *outputBuffer =
| DevExt->DiskChangeCount;
38915|         } else {
38916|             Irp->IoStatus.Status =
| CheckMediaLoaded(DeviceObject,Irp);
38917|         }
38918|
38919|         break;
38920|     }
38921|     case IOCTL_STORAGE_MEDIA_REMOVAL :
38922|     case IOCTL_DISK_MEDIA_REMOVAL: {
38923|         PPREVENT_MEDIA_REMOVAL Pmr =
| (PPREVENT_MEDIA_REMOVAL)Irp->AssociatedIrp.SystemBuffer;
38924|         #if DO_ALL_IO
38925|
| Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
| Ioctl=%08x - %s\n",
38926|
| DeviceObject,

```



```

38927|
    | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
    | e,
38928|
    | File_GetIOCTLString(currentIrpStack->Parameters.DeviceIo
    | Control.IoControlCode))););
38929|
    #endif /*DO_ALL_IO*/
38930|
    if (
    | currentIrpStack->Parameters.DeviceIoControl.InputBufferL
    | ength < sizeof( PREVENT_MEDIA_REMOVAL ) ) {
38931|
        Irp->IoStatus.Status =
    | STATUS_BUFFER_TOO_SMALL;
38932|
    } else {
38933|
38934|
        if (
    | Pmr->PreventMediaRemoval ) {
38935|
            // Prevent Removal
38936|
    | InterlockedIncrement((PLONG) &DevExt->LockCount);
38937|
    | Debug(DEBUG_DEVCON,("VDisk: Prevent Removal
    | %08x\n",DevExt->LockCount));
38938|
        Irp->IoStatus.Status =
    | STATUS_SUCCESS;
38939|
    } else {
38940|
        // Allow Removal
38941|
        if ( DevExt->LockCount
    | ) {
38942|
    | InterlockedDecrement((PLONG) &DevExt->LockCount);
38943|
    }
38944|
    | Debug(DEBUG_DEVCON,("VDisk: Allow Removal
    | %08x\n",DevExt->LockCount));
38945|
        Irp->IoStatus.Status =
    | STATUS_SUCCESS;
38946|
    }
38947|
    }
38948|
    break;
38949|
    }
38950|
    case IOCTL_DISK_IS_WRITABLE: {
38951|
        if (
    | CheckMediaLoaded(DeviceObject,Irp)==STATUS_SUCCESS ) {
38952|
            if ( gAllowWrites ) {
38953|
    | Debug(DEBUG_DEVCON,("VDisk: Not write protected\n"));
38954|
        Irp->IoStatus.Status =
    | STATUS_SUCCESS;
38955|
    } else {
38956|
        // now see if we made

```

```

    | the device read only (ie a fat drive)
38957|         if (
    | DevExt->DeviceObject->Characteristics &
    | FILE_READ_ONLY_DEVICE ) {
38958|         | Debug(DEBUG_DEVCON,("VDisk: Write Protected\n"));
38959|         | Irp->IoStatus.Status = STATUS_MEDIA_WRITE_PROTECTED;
38960|         } else {
38961|         | Debug(DEBUG_DEVCON,("VDisk: Not write protected\n"));
38962|         | Irp->IoStatus.Status = STATUS_SUCCESS;
38963|         }
38964|     }
38965| }
38966|     break;
38967| }
38968|     case IOCTL_STORAGE_GET_MEDIA_TYPES:
38969|     case IOCTL_DISK_GET_MEDIA_TYPES:
38970|     case IOCTL_DISK_GET_DRIVE_GEOMETRY: {
38971|
38972|         // Return the drive geometry
    | for the specified drive.
38973|         #if DO_ALL_IO
38974|         | Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
    | Ioctl=%08x - %s\n",
38975|         | DeviceObject,
38976|         | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
    | e,
38977|         | File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
    | Control.IoControlCode)));
38978|         #endif /*DO_ALL_IO*/
38979|         if (
    | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
    | Length < sizeof( DISK_GEOMETRY ) ) {
38980|             Irp->IoStatus.Status =
    | STATUS_BUFFER_TOO_SMALL;
38981|         } else {
38982|             PDISK_GEOMETRY
    | outputBuffer;
38983|
38984|             outputBuffer =
    | (PDISK_GEOMETRY)Irp->AssociatedIrp.SystemBuffer;
38985|             outputBuffer->MediaType =
    | RemovableMedia;

```

```

38986|
38987|             outputBuffer->Cylinders =
| DevExt->Cylinders;
38988|
| outputBuffer->TracksPerCylinder = DevExt->Heads;
38989|
| outputBuffer->SectorsPerTrack = DevExt->SPT;
38990|
| outputBuffer->BytesPerSector = DevExt->BPS;
38991|
38992|             Irp->IoStatus.Status =
| STATUS_SUCCESS;
38993|             Irp->IoStatus.Information =
| sizeof( DISK_GEOMETRY );
38994|         }
38995|
38996|         break;
38997|     }
38998|     case IOCTL_DISK_GET_DRIVE_LAYOUT: {
38999|         PIRP          newIrp=NULL;
39000|         IO_STATUS_BLOCK
| ioStatusBlock={0};
39001|         KEVENT        Event={0};
39002|         PDRIVE_LAYOUT_INFORMATION
| partitionInfo=
| (PDRIVE_LAYOUT_INFORMATION)Irp->AssociatedIrp.SystemBuffer;
| er;
39003|
39004|         if (
| CheckMediaLoaded(DeviceObject,Irp)==STATUS_SUCCESS ) {
39005|
39006|             //
39007|             // Perform the get
| partition info synchronously. Set both
39008|             // the input and output
| buffers as the buffer passed.
39009|             //
39010|
39011|             KeInitializeEvent(&Event,
| NotificationEvent, FALSE);
39012|
39013|             PFILTERED_EXTENSION
| FilteredExt = GetFilteredExtension(DevExt->PSMDDevice);
39014|
39015|             newIrp =
| IoBuildDeviceIoControlRequest (
39016|
| IOCTL_DISK_GET_DRIVE_LAYOUT,
39017|
| FilteredExt->TargetDeviceObject,

```

```

39018|                partitionInfo,
39019|
39019| | currentIrpStack->Parameters.DeviceIoControl.InputBufferL
39019| | ength,
39020|                partitionInfo,
39021|
39021| | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
39021| | Length,
39022|                FALSE,
39023|                &Event,
39024|                &ioStatusBlock );
39025|
39026|                Status = IoCallDriver
39026| | (FilteredExt->TargetDeviceObject, newIrp);
39027|
39028|                if ( Status ==
39028| | STATUS_PENDING ) {
39029|
39029| | pmWaitForSingleObject(&Event,NULL);
39030|                Status =
39030| | ioStatusBlock.Status;
39031|                }
39032|
39033|                if ( NT_SUCCESS(Status) ) {
39034|                ULONG partNumber;
39035|
39036|                Irp->IoStatus =
39036| | ioStatusBlock;
39037|
39037| | Debug(DEBUG_DEVCON,("VD: %d partitions
39037| | found\n",partitionInfo->PartitionCount));
39038|                for ( partNumber = 0;
39038| | partNumber < partitionInfo->PartitionCount;
39038| | partNumber++ ) {
39039|
39039| | Debug(DEBUG_DEVCON,("VD: '%S' %2d: Offset=%08x%08x,
39039| | Length=%08x%08x, Hidden=%08x, Type=%02x\n",
39040|
39040| | DevExt->Name,
39041|
39041| | partNumber,
39042|
39042| | partitionInfo->PartitionEntry[partNumber].StartingOffset
39042| | .HighPart,
39043|
39043| | partitionInfo->PartitionEntry[partNumber].StartingOffset
39043| | .LowPart,
39044|
39044| | partitionInfo->PartitionEntry[partNumber].PartitionLengt
39044| | h.HighPart,

```

```

39045|
| partitionInfo->PartitionEntry[partNumber].PartitionLengt
| h.LowPart,
39046|
| partitionInfo->PartitionEntry[partNumber].HiddenSectors,
39047|
| partitionInfo->PartitionEntry[partNumber].PartitionType
39048|
| ));
39049|                // since we filter
| on top of ftdisk, change the codes.
39050|                if (
| partitionInfo->PartitionEntry[partNumber].PartitionType=
| =0x86 ) {
39051|
| partitionInfo->PartitionEntry[partNumber].PartitionType
| = 0x06;
39052|                }
39053|                if (
| partitionInfo->PartitionEntry[partNumber].PartitionType=
| =0x87 ) {
39054|
| partitionInfo->PartitionEntry[partNumber].PartitionType
| = 0x07;
39055|                }
39056|                }
39057|
39058|                IoIncrement =
| IO_DISK_INCREMENT;
39059|                } else {
39060|
| Irp->IoStatus.Information = 0;
39061|                Irp->IoStatus.Status =
| Status;
39062|
| Debug(DEBUG_DEVCON,("VD: %08x getting disk partition
| info for layout=n",Status));
39063|                }
39064|                }
39065|                break;
39066|                }
39067|                case IOCTL_DISK_GET_PARTITION_INFO: {
39068|                PIRP                newIrp=NULL;
39069|                IO_STATUS_BLOCK
| ioStatusBlock={0};
39070|                KEVENT                event={0};
39071|                PPARTITION_INFORMATION
| partitionInfo=
| (PPARTITION_INFORMATION)Irp->AssociatedIrp.SystemBuffer;
39072|

```

```

39073|             if (
39074|                 | CheckMediaLoaded(DeviceObject,Irp)==STATUS_SUCCESS ) {
39075|                 //
39076|                 // Perform the get
39077|                 | partition info synchronously. Set both
39078|                 // the input and output
39079|                 | buffers as the buffer passed.
39080|                 //
39081|                 KelInitializeEvent(&event,
39082|                 | NotificationEvent, FALSE);
39083|                 PFILTERED_EXTENSION
39084|                 | FilteredExt = GetFilteredExtension (DevExt->PSMDevice);
39085|                 newIrp =
39086|                 | IoBuildDeviceIoControlRequest(IOCTL_DISK_GET_PARTITION_I
39087|                 | NFO,
39088|                 | FilteredExt->TargetDeviceObject,
39089|                 | partitionInfo,
39090|                 | currentIrpStack->Parameters.DeviceIoControl.InputBufferL
39091|                 | ength,
39092|                 | partitionInfo,
39093|                 | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
39094|                 | Length,
39095|                 | FALSE,
39096|                 | &event,
39097|                 | &ioStatusBlock);
39098|                 Status = IoCallDriver
39099|                 | (FilteredExt->TargetDeviceObject, newIrp);
39100|                 if ( Status ==
39101|                 | STATUS_PENDING ) {
39102|                 | pmWaitForSingleObject(&event,NULL);
39103|                 Status =
39104|                 | ioStatusBlock.Status;
39105|                 }
39106|                 if ( NT_SUCCESS(Status) ) {
39107|

```

```

39102|                Irp->IoStatus =
    | ioStatusBlock;
39103|
39104|    | Debug(DEBUG_DEVCON,("VD: '%S': Offset=%08x%08x,
    | Length=%08x%08x, Hidden=%08x, Type=%02x\n"
39105|                "
    | BI=%08x, RP=%08x, RW=%08x, PN=%08x\n",
39106|    | DevExt->Name,
39107|    | partitionInfo->StartingOffset.HighPart,
39108|    | partitionInfo->StartingOffset.LowPart,
39109|    | partitionInfo->PartitionLength.HighPart,
39110|    | partitionInfo->PartitionLength.LowPart,
39111|    | partitionInfo->HiddenSectors,
39112|    | partitionInfo->PartitionType,
39113|    | partitionInfo->BootIndicator,
39114|    | partitionInfo->RecognizedPartition,
39115|    | partitionInfo->RewritePartition,
39116|    | partitionInfo->PartitionNumber
39117|                ));
39118|
39119|                // since we filter on
    | top of ftdisk, change the codes.
39120|                if (
    | partitionInfo->PartitionType==0x86 ) {
39121|    | partitionInfo->PartitionType = 0x06;
39122|                }
39123|                if (
    | partitionInfo->PartitionType==0x87 ) {
39124|    | partitionInfo->PartitionType = 0x07;
39125|                }
39126|
39127|                IoIncrement =
    | IO_DISK_INCREMENT;
39128|                } else {
39129|    | Irp->IoStatus.Information = 0;

```

```

39130|                Irp->IoStatus.Status =
| Status;
39131|
| Debug(DEBUG_DEVCON,("VD: %08x getting disk partition
| info\n",Status));
39132|                }
39133|                }
39134|                break;
39135|                }
39136|                case IOCTL_DISK_INTERNAL_SET_VERIFY: {
39137|                // If the caller is kernel
| mode, set the verify bit.
39138|
39139|                if (
| (KPROCESSOR_MODE)Irp->RequestorMode ==
| (KPROCESSOR_MODE)KernelMode ) {
39140|                DeviceObject->Flags |=
| DO_VERIFY_VOLUME;
39141|                }
39142|                Irp->IoStatus.Status =
| STATUS_SUCCESS;
39143|                break;
39144|                }
39145|                case IOCTL_DISK_INTERNAL_CLEAR_VERIFY:
| {
39146|                // If the caller is kernel
| mode, clear the verify bit.
39147|
39148|                if (
| (KPROCESSOR_MODE)Irp->RequestorMode ==
| (KPROCESSOR_MODE)KernelMode ) {
39149|                DeviceObject->Flags &=
| ~DO_VERIFY_VOLUME;
39150|                }
39151|                Irp->IoStatus.Status =
| STATUS_SUCCESS;
39152|                break;
39153|                }
39154|
39155|                // this if for fixed media.. i dont
| think i need it..
39156|                case IOCTL_DISK_VERIFY: {
39157|                PVERIFY_INFORMATION
| verifyInformation;
39158|                //
39159|                // Move parameters from the
| VerifyInformation structure to
39160|                // the READ parameters area, so
| that we'll find them when
39161|                // we try to treat this like a

```



```

    | READ.
39162|          //
39163|
39164|          #if DO_ALL_IO
39165|
    | Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
    | Ioctl=%08x - %s\n",
39166|
    | DeviceObject,
39167|
    | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
    | e,
39168|
    | File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
    | Control.IoControlCode)));
39169|          #endif /*DO_ALL_IO*/
39170|          verifyInformation =
    | (_VERIFY_INFORMATION *)
    | Irp->AssociatedIrp.SystemBuffer;
39171|
39172|
    | currentIrpStack->Parameters.Read.ByteOffset.LowPart =
    | verifyInformation->StartingOffset.LowPart;
39173|
    | currentIrpStack->Parameters.Read.ByteOffset.HighPart =
    | verifyInformation->StartingOffset.HighPart;
39174|
    | currentIrpStack->Parameters.Read.Length =
    | verifyInformation->Length;
39175|
39176|          //
39177|          // A VERIFY is identical to a
    | READ, except for the fact that no
39178|          // data gets transferred. So
    | follow the READ code path.
39179|          //
39180|
39181|          ReleaseVDiskResource();
39182|          Status = PSManRead(
    | DeviceObject, Irp );
39183|          // so the __finally wont
    | crash..
39184|          AcquireVDiskResource();
39185|          // the Irp has been completed
    | already
39186|          try_return(CompleteRequest =
    | FALSE);
39187|          }
39188|          // VDisk Only device IOCTL's!!!!
39189|          case IOCTL_UNWRITE_PROTECT : {

```

```

39190|                #if DO_ALL_IO
39191|
39192|    | Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
39193|    | Ioctl=%08x - %s\n",
39194|    | DeviceObject,
39195|    | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
39196|    | e,
39197|    | File_GetIoctlString(currentIrpStack->Parameters.DeviceIo
39198|    | Control.IoControlCode)));
39199|                #endif /*DO_ALL_IO*/
39200|                if ( !gAllowWrites ) {
39201|
39202|    | DevExt->KeepWriteInMemory=1;
39203|
39204|    | DeviceObject->Characteristics &=
39205|    | ~FILE_READ_ONLY_DEVICE;
39206|                // tell the os that it
39207|    | something changed...
39208|                if (
39209|    | (DevExt->DeviceObject->Vpb) && (
39210|    | DevExt->DeviceObject->Vpb->Flags & VPB_MOUNTED ) ) {
39211|
39212|    | DevExt->DeviceObject->Flags |= DO_VERIFY_VOLUME;
39213|                }
39214|                }
39215|
39216|                Irp->IoStatus.Status =
39217|    | STATUS_SUCCESS;
39218|                break;
39219|                }
39220|                // VDisk Only device IOCTL's!!!!
39221|                case IOCTL_IS_PSM_VOLUME : {
39222|                // return success instead of
39223|    | failure..
39224|                Irp->IoStatus.Status =
39225|    | STATUS_SUCCESS;
39226|                break;
39227|                }
39228|                case IOCTL_WRITE_PROTECT : {
39229|                #if DO_ALL_IO
39230|
39231|    | Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk: Dev=%p,
39232|    | Ioctl=%08x - %s\n",
39233|    | DeviceObject,
39234|    | currentIrpStack->Parameters.DeviceIoControl.IoControlCod

```

```

    | e,
39219|
    | File_GetIOCTLString(currentIrpStack->Parameters.DeviceIo
    | Control.IoControlCode));
39220|         #endif /*DO_ALL_IO*/
39221|         if ( !gAllowWrites ) {
39222|
    | DevExt->KeepWriteInMemory=0;
39223|         if (
    | DevExt->OriginalWriteProtected ) {
39224|
    | DeviceObject->Characteristics |= FILE_READ_ONLY_DEVICE;
39225|         // tell the os that it
    | something changed...
39226|         if (
    | (DevExt->DeviceObject->Vpb) && (
    | DevExt->DeviceObject->Vpb->Flags & VPB_MOUNTED ) ) {
39227|
    | DevExt->DeviceObject->Flags |= DO_VERIFY_VOLUME;
39228|         }
39229|     }
39230| }
39231|     Irp->IoStatus.Status =
    | STATUS_SUCCESS;
39232|     break;
39233| }
39234| #if _WIN32_WINNT >= 0x0500
39235| #ifndef PBYTE
39236| #define PBYTE unsigned char *
39237| #endif
39238|
39239|     case CTL_CODE(IOCTL_DISK_BASE, 0x0012,
    | METHOD_BUFFERED, FILE_READ_ACCESS):
39240|     case IOCTL_DISK_GET_PARTITION_INFO_EX
    | : {
39241|         PIRP         newIrp=NULL;
39242|         IO_STATUS_BLOCK
    | ioStatusBlock={0};
39243|         KEVENT         event={0};
39244|         PPARTITION_INFORMATION_EX
    | Info=(PPARTITION_INFORMATION_EX)Irp->AssociatedIrp.Syste
    | mBuffer;
39245|
39246|         if (
    | CheckMediaLoaded(DeviceObject,Irp)==STATUS_SUCCESS ) {
39247|             //
39248|             // Perform the get
    | partition info synchronously. Set both
39249|             // the input and output
    | buffers as the buffer passed.

```

```

39250|                //
39251|
39252|                KeInitializeEvent(&event,
    | NotificationEvent, FALSE);
39253|
39254|                PFILTERED_EXTENSION
    | FilteredExt = GetFilteredExtension (DevExt->PSMDevice);
39255|
39256|                newIrp =
    | IoBuildDeviceIoControlRequest (
39257|
    | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
    | e,
39258|
    | FilteredExt->TargetDeviceObject,
39259|
    | Irp->AssociatedIrp.SystemBuffer,
39260|
    | currentIrpStack->Parameters.DeviceIoControl.InputBufferL
    | ength,
39261|
    | Irp->AssociatedIrp.SystemBuffer,
39262|
    | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
    | Length,
39263|
    | FALSE,
39264|
    | &event,
39265|
    | &ioStatusBlock );
39266|
39267|                Status = IoCallDriver
    | (FilteredExt->TargetDeviceObject, newIrp);
39268|
39269|                if ( Status ==
    | STATUS_PENDING ) {
39270|
    | pmWaitForSingleObject(&event,NULL);
39271|
    | Status =
    | ioStatusBlock.Status;
39272|
    | }
39273|
39274|                if ( NT_SUCCESS(Status) ) {
39275|
    | Debug(DEBUG_DEVCON,("VD: Success sending ioctl\n"));
39276|
    | Irp->IoStatus =
    | ioStatusBlock;
39277|
    | IoIncrement =
    | IO_DISK_INCREMENT;
39278|
39279|
    | Debug(DEBUG_DEVCON,("VD: GetPart: Ps=%08x, So=%l64x,

```

```

    | L=%l64x, pn=%d, rp=%d\n",
39280|
    | Info->PartitionStyle,
39281|
    | Info->StartingOffset,
39282|
    | Info->PartitionLength,
39283|
    | Info->PartitionNumber,
39284|
    | Info->RewritePartition
39285|                                     ));
39286|                                     if (
    | Info->PartitionStyle==PARTITION_STYLE_MBR ) {
39287|
    | Debug(DEBUG_DEVCON,("VD:      : Pt=%02x, bi=%d,
    | rp=%d, hidden=%08x\n",
39288|
    | Info->Mbr.PartitionType,
39289|
    | Info->Mbr.BootIndicator,
39290|
    | Info->Mbr.RecognizedPartition,
39291|
    | Info->Mbr.HiddenSectors
39292|
    | ));
39293|                                     } else
39294|                                     if (
    | Info->PartitionStyle==PARTITION_STYLE_GPT ) {
39295|                                     UNICODE_STRING PT;
39296|                                     UNICODE_STRING ID;
39297|
    | RtlStringFromGUID(Info->Gpt.PartitionType,&PT);
39298|
    | RtlStringFromGUID(Info->Gpt.PartitionId,&ID);
39299|
39300|
    | Debug(DEBUG_DEVCON,("VD:      : Pt=%wZ, Id=%wZ,
    | A=%l64x, Name='%S'\n",
39301|
    | &PT,
39302|
    | &ID,
39303|
    | Info->Gpt.Attributes,
39304|
    | Info->Gpt.Name
39305|
    | ));

```

```

39306|
    | RtlFreeUnicodeString(&PT);
39307|
    | RtlFreeUnicodeString(&ID);
39308|                } else {
39309|
    | Debug(DEBUG_DEVCON,("VD: Unknown partition type\n"));
39310|                }
39311|                } else {
39312|
    | Irp->IoStatus.Information = 0;
39313|                Irp->IoStatus.Status =
    | Status;
39314|
    | Debug(DEBUG_DEVCON,("VD: %08x sending
    | ioctl\n",Status));
39315|                }
39316|                }
39317|                break;
39318|                }
39319|                case CTL_CODE(IOCTL_DISK_BASE, 0x0014,
    | METHOD_BUFFERED, FILE_READ_ACCESS):
39320|                case IOCTL_DISK_GET_DRIVE_LAYOUT_EX : {
39321|                PIRP        newIrp=NULL;
39322|                IO_STATUS_BLOCK
    | ioStatusBlock={0};
39323|                KEVENT        event={0};
39324|                PDRIVE_LAYOUT_INFORMATION_EX
    | Info=(PDRIVE_LAYOUT_INFORMATION_EX)Irp->AssociatedIrp.Sy
    | stemBuffer;
39325|
39326|                if (
    | CheckMediaLoaded(DeviceObject,Irp)==STATUS_SUCCESS ) {
39327|                //
39328|                // Perform the get
    | partition info synchronously. Set both
39329|                // the input and output
    | buffers as the buffer passed.
39330|                //
39331|
39332|                KeInitializeEvent(&event,
    | NotificationEvent, FALSE);
39333|
39334|                PFILTERED_EXTENSION
    | FilteredExt = GetFilteredExtension (DevExt->PSMDevice);
39335|
39336|                newIrp =
    | IoBuildDeviceIoControlRequest(
39337|                | currentIrpStack->Parameters.DeviceIoControl.IoControlCod

```

```

    | e,
39338|
    | FilteredExt->TargetDeviceObject,
39339|
    | Irp->AssociatedIrp.SystemBuffer,
39340|
    | currentIrpStack->Parameters.DeviceIoControl.InputBufferL
    | ength,
39341|
    | Irp->AssociatedIrp.SystemBuffer,
39342|
    | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
    | Length,
39343|                FALSE,
39344|                &event,
39345|                &ioStatusBlock);
39346|
39347|                Status = IoCallDriver
    | (FilteredExt->TargetDeviceObject, newIrp);
39348|
39349|                if ( Status ==
    | STATUS_PENDING ) {
39350|
    | pmWaitForSingleObject(&event,NULL);
39351|                Status =
    | ioStatusBlock.Status;
39352|                }
39353|
39354|                if ( NT_SUCCESS(Status) ) {
39355|
    | Debug(DEBUG_DEVCON,("VD: Success sending ioctl\n"));
39356|                Irp->IoStatus =
    | ioStatusBlock;
39357|                IoIncrement =
    | IO_DISK_INCREMENT;
39358|
39359|
    | Debug(DEBUG_DEVCON,("Vd: GetDriveLayout: PS=%08x,
    | pc=%08x\n",Info->PartitionStyle,
    | Info->PartitionCount));
39360|                if (
    | Info->PartitionStyle==PARTITION_STYLE_MBR ) {
39361|
    | Debug(DEBUG_DEVCON,("Vd: MBR:
    | Sig=%08x\n",Info->Mbr.Signature));
39362|                } else
39363|                if (
    | Info->PartitionStyle==PARTITION_STYLE_GPT ) {
39364|                UNICODE_STRING GS;
39365|

```

```

    | RtlStringFromGUID(Info->Gpt.DiskId,&GS);
39366|
39367|
    | Debug(DEBUG_DEVCON,("Vd: MBR: Guid=%08x, so=%l64x,
    | l=%l64x,
    | count=%08x\n",&GS,Info->Gpt.StartingUsableOffset,Info->G
    | pt.UsableLength,Info->Gpt.MaxPartitionCount));
39368|
39369|
    | RtlFreeUnicodeString(&GS);
39370|                } else {
39371|
    | Debug(DEBUG_DEVCON,("VD: GetDriveLayout: Unknown
    | %08x\n",Info->PartitionStyle));
39372|                }
39373|
39374|                for ( ULONG
    | i=0;i<Info->PartitionCount;i++ ) {
39375|
    | Debug(DEBUG_DEVCON,("VD: Part[%2d]: Ps=%08x, So=%l64x,
    | L=%l64x, pn=%d, rp=%d\n",
39376|
    | i,
39377|
    | Info->PartitionEntry[i].PartitionStyle,
39378|
    | Info->PartitionEntry[i].StartingOffset,
39379|
    | Info->PartitionEntry[i].PartitionLength,
39380|
    | Info->PartitionEntry[i].PartitionNumber,
39381|
    | Info->PartitionEntry[i].RewritePartition
39382|
    | ));
39383|                if (
    | Info->PartitionEntry[i].PartitionStyle==PARTITION_STYLE_
    | MBR ) {
39384|
    | Debug(DEBUG_DEVCON,("VD:      : Pt=%02x, bi=%d,
    | rp=%d, hidden=%08x\n",
39385|
    | Info->PartitionEntry[i].Mbr.PartitionType,
39386|
    | Info->PartitionEntry[i].Mbr.BootIndicator,
39387|
    | Info->PartitionEntry[i].Mbr.RecognizedPartition,
39388|
    | Info->PartitionEntry[i].Mbr.HiddenSectors
39389|

```



```

| ));
39390|                } else
39391|                if (
| Info->PartitionEntry[i].PartitionStyle==PARTITION_STYLE_
| GPT ) {
39392|                UNICODE_STRING
| PT;
39393|                UNICODE_STRING
| ID;
39394|
| RtlStringFromGUID(Info->PartitionEntry[i].Gpt.PartitionT
| ype,&PT);
39395|
| RtlStringFromGUID(Info->PartitionEntry[i].Gpt.PartitionI
| d,&ID);
39396|
39397|
| Debug(DEBUG_DEVCON,("VD:      : Pt=%wZ, Id=%wZ,
| A=%!64x, Name='%S'\n",
39398|
| &PT,
39399|
| &ID,
39400|
| Info->PartitionEntry[i].Gpt.Attributes,
39401|
| Info->PartitionEntry[i].Gpt.Name
39402|
| ));
39403|
| RtlFreeUnicodeString(&PT);
39404|
| RtlFreeUnicodeString(&ID);
39405|                } else {
39406|
| Debug(DEBUG_DEVCON,("VD: Unknown partition type\n"));
39407|                }
39408|                }
39409|                } else {
39410|
| Irp->IoStatus.Information = 0;
39411|                Irp->IoStatus.Status =
| Status;
39412|
| Debug(DEBUG_DEVCON,("VD: %08x sending
| ioctl\n",Status));
39413|                }
39414|                }
39415|                break;
39416|                }

```

```

39417|         case IOCTL_DISK_GET_LENGTH_INFO : {
39418|             PIRP             newIrp=NULL;
39419|             IO_STATUS_BLOCK
39420|             | ioStatusBlock={0};
39421|             KEVENT             event={0};
39422|             PGET_LENGTH_INFORMATION
39423|             | Len=(PGET_LENGTH_INFORMATION)Irp->AssociatedIrp.SystemBu
39424|             | ffer;
39425|             if (
39426|             | CheckMediaLoaded(DeviceObject,Irp)==STATUS_SUCCESS ) {
39427|                 //
39428|                 // Perform the get
39429|                 | partition info synchronously. Set both
39430|                 // the input and output
39431|                 | buffers as the buffer passed.
39432|                 //
39433|                 KeInitializeEvent(&event,
39434|                 | NotificationEvent, FALSE);
39435|                 PFILTERED_EXTENSION
39436|                 | FilteredExt = GetFilteredExtension (DevExt->PSMDevice);
39437|                 newIrp =
39438|                 | IoBuildDeviceIoControlRequest (
39439|                 | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
39440|                 | e,
39441|                 | FilteredExt->TargetDeviceObject,
39442|                 | Irp->AssociatedIrp.SystemBuffer,
39443|                 | currentIrpStack->Parameters.DeviceIoControl.InputBufferL
39444|                 | ength,
39445|                 | Irp->AssociatedIrp.SystemBuffer,
39446|                 | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
39447|                 | Length,
39448|                 FALSE,
39449|                 &event,
39450|                 &ioStatusBlock );
39451|                 Status = IoCallDriver
39452|                 | (FilteredExt->TargetDeviceObject, newIrp);
39453|                 if ( Status ==
39454|                 | STATUS_PENDING ) {

```

```

39447|
| pmWaitForSingleObject(&event,NULL);
39448|             Status =
| ioStatusBlock.Status;
39449|         }
39450|
39451|             if ( NT_SUCCESS(Status) ) {
39452|
| Debug(DEBUG_DEVCON,("VD: Success sending ioctl\n"));
39453|             Irp->IoStatus =
| ioStatusBlock;
39454|             IoIncrement =
| IO_DISK_INCREMENT;
39455|
39456|             Debug(DEBUG_DEVCON,("Vd: GetLength:
| Length=%I64x\n",Len->Length));
39457|             } else {
39458|
| Irp->IoStatus.Information = 0;
39459|             Irp->IoStatus.Status =
| Status;
39460|
| Debug(DEBUG_DEVCON,("VD: %08x sending
| ioctl\n",Status));
39461|             }
39462|         }
39463|         break;
39464|     }
39465|     case CTL_CODE(IOCTL_DISK_BASE, 0x0028,
| METHOD_BUFFERED, FILE_READ_ACCESS):
39466|     case IOCTL_DISK_GET_DRIVE_GEOMETRY_EX :
| {
39467|         PIRP         newIrp=NULL;
39468|         IO_STATUS_BLOCK
| ioStatusBlock={0};
39469|         KEVENT         event={0};
39470|         PDISK_GEOMETRY_EX
| Geo=(PDISK_GEOMETRY_EX)Irp->AssociatedIrp.SystemBuffer;
39471|
39472|         if (
| CheckMediaLoaded(DeviceObject,Irp)==STATUS_SUCCESS ) {
39473|             //
39474|             // Perform the get
| partition info synchronously. Set both
39475|             // the input and output
| buffers as the buffer passed.
39476|             //
39477|
39478|             KeInitializeEvent(&event,

```

```

    | NotificationEvent, FALSE);
39479|
39480|             PFILTERED_EXTENSION
    | FilteredExt = GetFilteredExtension (DevExt->PSMDevice);
39481|
39482|             newIrp =
    | IoBuildDeviceIoControlRequest (
39483|
    | currentIrpStack->Parameters.DeviceIoControl.IoControlCod
    | e,
39484|
    | FilteredExt->TargetDeviceObject,
39485|
    | Irp->AssociatedIrp.SystemBuffer,
39486|
    | currentIrpStack->Parameters.DeviceIoControl.InputBufferL
    | ength,
39487|
    | Irp->AssociatedIrp.SystemBuffer,
39488|
    | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
    | Length,
39489|             FALSE,
39490|             &event,
39491|             &ioStatusBlock);
39492|
39493|             Status = IoCallDriver
    | (FilteredExt->TargetDeviceObject, newIrp);
39494|
39495|             if ( Status ==
    | STATUS_PENDING ) {
39496|
    | pmWaitForSingleObject(&event,NULL);
39497|             Status =
    | ioStatusBlock.Status;
39498|             }
39499|
39500|             if ( NT_SUCCESS(Status) ) {
39501|                 PDISK_PARTITION_INFO
    | Info;
39502|                 PDISK_DETECTION_INFO
    | Detect;
39503|
39504|
    | Debug(DEBUG_DEVCON,("VD: Success sending ioctl\n"));
39505|             Irp->IoStatus =
    | ioStatusBlock;
39506|             IoIncrement =
    | IO_DISK_INCREMENT;
39507|

```

```

39508|
    | Debug(DEBUG_DEVCON,("VD: Geometry: Cyls=%l64d, MT=%08x,
    | TPC=%d, SPT=%d, BPS=%d\n",
39509|
    | Geo->Geometry.Cylinders,
39510|
    | Geo->Geometry.MediaType,
39511|
    | Geo->Geometry.TracksPerCylinder,
39512|
    | Geo->Geometry.SectorsPerTrack,
39513|
    | Geo->Geometry.BytesPerSector));
39514|
39515|
    | Info=DiskGeometryGetPartition(Geo);
39516|
    | Detect=DiskGeometryGetDetect(Geo);
39517|
39518|                if (
    | Info->PartitionStyle==PARTITION_STYLE_MBR ) {
39519|
    | Debug(DEBUG_DEVCON,("VD: Info: PS=%08x (MBR), Sig=%08x,
    | Checksum=%08x\n",
39520|
    | Info->PartitionStyle,
    | Info->Mbr.Signature,Info->Mbr.CheckSum));
39521|                } else
39522|                if (
    | Info->PartitionStyle==PARTITION_STYLE_GPT ) {
39523|                UNICODE_STRING GS;
39524|
    | RtlStringFromGUID(Info->Gpt.DiskId,&GS);
39525|
39526|
    | Debug(DEBUG_DEVCON,("VD: Info: PS=%08x (GPT),
    | Guid=%wZ\n",
39527|
    | Info->PartitionStyle, &GS));
39528|
39529|
    | RtlFreeUnicodeString(&GS);
39530|                } else
39531|                if (
    | Info->PartitionStyle==PARTITION_STYLE_RAW ) {
39532|
    | Debug(DEBUG_DEVCON,("VD: Info: PS=%08x (RAW)\n",
39533|
    | Info->PartitionStyle));
39534|                } else {

```

```

39535|
| Debug(DEBUG_DEVCON,("VD: Info: PS=%08x (Unknown)\n",
39536|
| Info->PartitionStyle));
39537|
}
39538|
39539|
if (
| Detect->DetectionType==DetectNone ) {
39540|
| Debug(DEBUG_DEVCON,("VD: Detect: %08x
| (None)\n",Detect->DetectionType));
39541|
} else
39542|
if (
| Detect->DetectionType==DetectInt13 ) {
39543|
| Debug(DEBUG_DEVCON,("VD: Detect: %08x (Int13) ds=%04x,
| Cyls=%08x, SPT=%04x, H=%04x, nd=%04x\n",
39544|
| Detect->DetectionType,
39545|
| Detect->Int13.DriveSelect,
39546|
| Detect->Int13.MaxCylinders,
39547|
| Detect->Int13.SectorsPerTrack,
39548|
| Detect->Int13.MaxHeads,
39549|
| Detect->Int13.NumberDrives
39550|
| ));
39551|
} else
39552|
if (
| Detect->DetectionType==DetectExInt13 ) {
39553|
| Debug(DEBUG_DEVCON,("VD: Detect: %08x (ExInt13)
| bs=%04x, f=%04x, C=%08x, h=%08x, spt=%08x, spd=%l64x,
| ss=%04x, r=%04x\n",
39554|
| Detect->DetectionType,
39555|
| Detect->ExInt13.ExBufferSize,
39556|
| Detect->ExInt13.ExFlags,
39557|
| Detect->ExInt13.ExCylinders,
39558|
| Detect->ExInt13.ExHeads,
39559|
| Detect->ExInt13.ExSectorsPerTrack,

```

```

39560|
| Detect->ExInt13.ExSectorsPerDrive,
39561|
| Detect->ExInt13.ExSectorSize,
39562|
| Detect->ExInt13.ExReserved
39563|
| ));
39564|                } else {
39565|
| Debug(DEBUG_DEVCON,("VD: Detect: %08x
| (Unknown)\n",Detect->DetectionType));
39566|                }
39567|                } else {
39568|
| Irp->IoStatus.Information = 0;
39569|                Irp->IoStatus.Status =
| Status;
39570|
| Debug(DEBUG_DEVCON,("VD: %08x sending
| ioctl\n",Status));
39571|                }
39572|                }
39573|                break;
39574|                }
39575|
39576|                case IOCTL_MOUNTDEV_QUERY_DEVICE_NAME :
| {
39577|                PMOUNTDEV_NAME Name =
| (PMOUNTDEV_NAME)Irp->AssociatedIrp.SystemBuffer;
39578|                WCHAR Buffer[200];
39579|                ULONG Len;
39580|
39581|                | sprintf(Buffer,L"\\Device\\PsmDevices_%04x\\%s_%d",PSM_
| LOW_COMPATIBLE_VERSION,DevExt->Name,DevExt->Instance);
39582|                Len =
| wcslen(Buffer)*sizeof(WCHAR);
39583|
39584|                ASSERT(wcslen(Buffer)<200);
39585|                if (
| currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
| Length >= sizeof( MOUNTDEV_NAME ) ) {
39586|                Name->NameLength =
| (USHORT)Len;
39587|
39588|                if (
| currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
| Length >= sizeof( MOUNTDEV_NAME )+Len ) {
39589|

```

```

    | RtlCopyMemory(Name->Name,Buffer,Len);
39590|
    | Irp->IoStatus.Information = FIELD_OFFSET(MOUNTDEV_NAME,
    | Name) +Len;
39591|         } else {
39592|         Irp->IoStatus.Status =
    | STATUS_BUFFER_OVERFLOW;
39593|
    | Irp->IoStatus.Information = sizeof(MOUNTDEV_NAME);
39594|         }
39595|     } else {
39596|         // too small for anything..
39597|         Irp->IoStatus.Status =
    | STATUS_BUFFER_TOO_SMALL;
39598|     }
39599|
39600|     break;
39601| }
39602| case
    | IOCTL_MOUNTDEV_QUERY_SUGGESTED_LINK_NAME: {
39603|     PMOUNTDEV_SUGGESTED_LINK_NAME
    | Name=(PMOUNTDEV_SUGGESTED_LINK_NAME)Irp->AssociatedIrp.S
    | ystemBuffer;
39604|     WCHAR Buffer[200];
39605|     ULONG Len;
39606|
39607|     | swprintf(Buffer,L"%s_%d",DevExt->Name,DevExt->Instance);
39608|     Len =
    | wcslen(Buffer)*sizeof(WCHAR);
39609|     if (
    | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
    | Length >= sizeof( MOUNTDEV_SUGGESTED_LINK_NAME) ) {
39610|
    | Name->UseOnlyIfThereAreNoOtherLinks = 1;
39611|     Name->NameLength =
    | (USHORT)Len;
39612|     if (
    | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
    | Length >= sizeof( MOUNTDEV_SUGGESTED_LINK_NAME)+Len ) {
39613|
    | RtlCopyMemory(Name->Name,Buffer,Len);
39614|
    | Irp->IoStatus.Information =
    | FIELD_OFFSET(MOUNTDEV_SUGGESTED_LINK_NAME,Name)+Len;
39615|         } else {
39616|         Irp->IoStatus.Status =
    | STATUS_BUFFER_OVERFLOW;
39617|
    | Irp->IoStatus.Information =

```



```

    | sizeof(MOUNTDEV_SUGGESTED_LINK_NAME);
39618|         }
39619|     } else {
39620|         Irp->IoStatus.Status =
    | STATUS_BUFFER_TOO_SMALL;
39621|     }
39622|     break;
39623| }
39624|     case IOCTL_MOUNTDEV_QUERY_UNIQUE_ID: {
39625|         PMOUNTDEV_UNIQUE_ID
    | Name=(PMOUNTDEV_UNIQUE_ID)Irp->AssociatedIrp.SystemBuffer
    | r;
39626|         WCHAR Buffer[200];
39627|         ULONG Len;
39628|
39629|         | swprintf(Buffer,L"%s_%d",DevExt->Name,DevExt->Instance);
39630|         Len =
    | wcslen(Buffer)*sizeof(WCHAR);
39631|         if (
    | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
    | Length >= sizeof( MOUNTDEV_UNIQUE_ID) ) {
39632|             Name->UniqueIdLength =
    | (USHORT)Len;
39633|             if (
    | currentIrpStack->Parameters.DeviceIoControl.OutputBuffer
    | Length >= sizeof( MOUNTDEV_UNIQUE_ID)+Len ) {
39634|                 | RtlCopyMemory(Name->UniqueId,Buffer,Len);
39635|                 | Irp->IoStatus.Information =
    | FIELD_OFFSET(MOUNTDEV_UNIQUE_ID,UniqueId)+Len;
39636|             } else {
39637|                 Irp->IoStatus.Status =
    | STATUS_BUFFER_OVERFLOW;
39638|                 | Irp->IoStatus.Information = sizeof(MOUNTDEV_UNIQUE_ID);
39639|             }
39640|         } else {
39641|             Irp->IoStatus.Status =
    | STATUS_BUFFER_TOO_SMALL;
39642|         }
39643|         break;
39644|     }
39645| #endif
39646|
39647|
39648|         // currently not supported ioctls
39649|
39650|         case IOCTL_SCSI_PASS_THROUGH:

```

```

39651|         case IOCTL_SCSI_MINIPORT:
39652|         case IOCTL_SCSI_GET_INQUIRY_DATA:
39653|         case IOCTL_SCSI_GET_CAPABILITIES:
39654|         case IOCTL_SCSI_PASS_THROUGH_DIRECT:
39655|         case IOCTL_SCSI_GET_ADDRESS:
39656|         case IOCTL_SCSI_RESCAN_BUS:
39657|         case IOCTL_SCSI_GET_DUMP_POINTERS:
39658|         case IOCTL_STORAGE_FIND_NEW_DEVICES:
39659|         case IOCTL_DISK_FIND_NEW_DEVICES:
39660| //         case IOCTL_DISK_REMOVE_DEVICE :
39661|         case IOCTL_DISK_CONTROLLER_NUMBER:
39662|             // fall through...
39663|         default:
39664|             Debug(DEBUG_DEVCON,("VDisk: ioctl
| not handled! %08x - %s\n",
39665|             | currentIrpStack->Parameters.DeviceIoControl.IoControlCode
| e,
39666|             | File_GetIOCTLString(currentIrpStack->Parameters.DeviceIo
| Control.IoControlCode)));
39667|             Irp->IoStatus.Status =
| STATUS_INVALID_DEVICE_REQUEST;
39668|
39669|         }
39670|     }
| __except(ExceptionFilter(GetExceptionInformation())) {
39671|         Status = GetExceptionCode();
39672|         Debug(DEBUG_DEVCON,("VDisk: Exception %08x
| in PSMANDeviceControlVDisk\n",Status));
39673|     }
39674|     try_exit: NOTHING;
39675| } __finally {
39676|     ReleaseVDiskResource();
39677| }
39678|
39679| if ( CompleteRequest ) {
39680|     // Finish the I/O operation by simply
| completing the packet and returning
39681|     // the same status as in the packet itself.
39682|
39683|     Status = Irp->IoStatus.Status;
39684|
39685|     if ( Status!=0 ) {
39686|         Debug(DEBUG_READ,("VDisk: devcon: Device
| %08x Irp %08x Error
| %08x\n",DevExt->DeviceObject,Irp,Status));
39687|     }
39688|     IoCompleteRequest( Irp, IoIncrement );
39689| }

```

```

39690|
39691| //Debug(DEBUG_DEVCON,("PSManDeviceControlVDisk:
    | Done Status=%08x\n",Status));
39692| return Status;
39693|
39694| }
39695|
39696| /*-----
    | -----*/
39697| STATIC NTSTATUS PSManDeviceControlFSObject(
39698|
    | PDEVICE_OBJECT DeviceObject,
39699|                                PIRP Irp
39700|                                )
39701| {
39702|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
39703|
39704|     Debug(DEBUG_PROCCALL |
    | DEBUG_CLEANUP,("PSManDevConFSObject Called Dev=%p,
    | Irp=%p\n",DeviceObject,Irp));
39705|     Irp->IoStatus.Information = 0;
39706|     Irp->IoStatus.Status = Status;
39707|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
39708|     Debug(DEBUG_PROCCALL |
    | DEBUG_CLEANUP,("PSManDevConFSObject Done\n"));
39709|
39710|     return Status;
39711|
39712| }
39713|
39714| /*-----
    | -----*/
39715| STATIC NTSTATUS PSManDeviceControlFSFilter(
39716|
    | PDEVICE_OBJECT DeviceObject,
39717|                                PIRP Irp
39718|                                )
39719| {
39720|     NTSTATUS Status;
39721|
39722| #ifdef DEBUG
39723|     if ( PsmActive ) {
39724|         Debug(DEBUG_PROCCALL |
    | DEBUG_PROCCALL,("PSManDevConFSFilter Called Device=%p,
    | Irp=%p\n",DeviceObject,Irp));
39725|     }
39726| #endif
39727|
39728|     Status = PSManFSPassThru( DeviceObject, Irp );
39729|

```

```

39730| #ifdef DEBUG
39731|     if ( PsmActive ) {
39732|         Debug(DEBUG_CLEANUP |
39733|             | DEBUG_PROCCALL,("PManDevConFSFilter Done   Device=%p,
39734|             | Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
39735|     }
39736| #endif
39737|     return Status;
39738| }
39739|
39740|
39741| File Listing: DEVCON.h
39742|
39743| STATIC NTSTATUS
39744| PManDeviceControlObject(
39745|     PDEVICE_OBJECT DeviceObject,
39746|     PIRP Irp
39747| );
39748|
39749| STATIC NTSTATUS
39750| PManDeviceControlDevice(
39751|     PDEVICE_OBJECT DeviceObject,
39752|     PIRP Irp
39753| );
39754|
39755| NTSTATUS
39756| PManDeviceControl(
39757|     PDEVICE_OBJECT DeviceObject,
39758|     PIRP Irp
39759| );
39760|
39761| STATIC NTSTATUS
39762| PManDeviceControlVdisk(
39763|     PDEVICE_OBJECT DeviceObject,
39764|     PIRP Irp
39765| );
39766|
39767| STATIC NTSTATUS
39768| PManDeviceControlFSObject(
39769|     PDEVICE_OBJECT DeviceObject,
39770|     PIRP Irp
39771| );
39772|
39773| STATIC NTSTATUS
39774| PManDeviceControlFSFilter(
39775|     PDEVICE_OBJECT DeviceObject,
39776|     PIRP Irp
39777| );

```

```

39778|
39779|
39780| STATIC NTSTATUS
39781| PSMANNewDiskCompletion(
39782|     IN PDEVICE_OBJECT DeviceObject,
39783|     IN PIRP           Irp,
39784|     IN PVOID          Context
39785| );
39786|
39787| NTSTATUS NotifyUserModeOfEvent( ULONG Event );
39788| NTSTATUS NotifyUserModeOfRegChangeEvent(
    | PFILTERED_EXTENSION FiltExt );
39789| NTSTATUS NotifyUserModeOfVolumeOnlineEvent(
    | PFILTERED_EXTENSION FiltExt );
39790|
39791|
39792| #if _WIN32_WINNT >= 0x0500
39793|
39794| // Ioctl's I got from build 2416 Whistler SDK (Post beta
    | 1) ntdddisk.h
39795| //
39796| // New IOCTLs for GUID Partition table disks.
39797| //
39798|
39799| #define IOCTL_DISK_GET_PARTITION_INFO_EX
    | CTL_CODE(IOCTL_DISK_BASE, 0x0012, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
39800| #define IOCTL_DISK_SET_PARTITION_INFO_EX
    | CTL_CODE(IOCTL_DISK_BASE, 0x0013, METHOD_BUFFERED,
    | FILE_READ_ACCESS | FILE_WRITE_ACCESS)
39801| #define IOCTL_DISK_GET_DRIVE_LAYOUT_EX
    | CTL_CODE(IOCTL_DISK_BASE, 0x0014, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
39802| #define IOCTL_DISK_SET_DRIVE_LAYOUT_EX
    | CTL_CODE(IOCTL_DISK_BASE, 0x0015, METHOD_BUFFERED,
    | FILE_READ_ACCESS | FILE_WRITE_ACCESS)
39803| #define IOCTL_DISK_CREATE_DISK
    | CTL_CODE(IOCTL_DISK_BASE, 0x0016, METHOD_BUFFERED,
    | FILE_READ_ACCESS | FILE_WRITE_ACCESS)
39804| #define IOCTL_DISK_GET_LENGTH_INFO
    | CTL_CODE(IOCTL_DISK_BASE, 0x0017, METHOD_BUFFERED,
    | FILE_READ_ACCESS)
39805| #define IOCTL_DISK_GET_DRIVE_GEOMETRY_EX
    | CTL_CODE(IOCTL_DISK_BASE, 0x0028, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
39806|
39807|
39808| //
39809| // Support for GUID Partition Table (GPT) disks.
39810| //

```

```

39811|
39812| //
39813| // There are currently two ways a disk can be
    | partitioned. With a traditional
39814| // AT-style master boot record (PARTITION_STYLE_MBR)
    | and with a new, GPT
39815| // partition table (PARTITION_STYLE_GPT). RAW is for an
    | unrecognizable
39816| // partition style. There are a very limited number of
    | things you can
39817| // do with a RAW partition.
39818| //
39819|
39820| typedef enum _PARTITION_STYLE {
39821|     PARTITION_STYLE_MBR,
39822|     PARTITION_STYLE_GPT,
39823|     PARTITION_STYLE_RAW
39824| } PARTITION_STYLE;
39825|
39826|
39827| //
39828| // The following structure defines information in a GPT
    | partition that is
39829| // not common to both GPT and MBR partitions.
39830| //
39831|
39832| typedef struct _PARTITION_INFORMATION_GPT {
39833|     GUID PartitionType;           // Partition
    | type. See table 16-3.
39834|     GUID PartitionId;           // Unique GUID
    | for this partition.
39835|     ULONG64 Attributes;         // See table
    | 16-4.
39836|     WCHAR Name [36];           // Partition
    | Name in Unicode.
39837| } PARTITION_INFORMATION_GPT,
    | *PPARTITION_INFORMATION_GPT;
39838|
39839| //
39840| // The following are GPT partition attributes
    | applicable when the
39841| // PartitionType is PARTITION_BASIC_DATA_GUID.
39842| //
39843|
39844| #define GPT_BASIC_DATA_ATTRIBUTE_NO_DRIVE_LETTER
    | (0x8000000000000000)
39845| #define GPT_BASIC_DATA_ATTRIBUTE_HIDDEN
    | (0x4000000000000000)
39846| #define GPT_BASIC_DATA_ATTRIBUTE_HIDDEN_IF_CLONE
    | (0x2000000000000000)

```

```

39847| #define GPT_BASIC_DATA_ATTRIBUTE_READ_ONLY
39848| | (0x10000000000000000)
39849| #define GPT_BASIC_DATA_ATTRIBUTE_READ_ONLY_IF_CLONE
39850| | (0x0800000000000000)
39851| // The following structure defines information in an
39852| // MBR partition that is not
39853| // common to both GPT and MBR partitions.
39854| //
39855| typedef struct _PARTITION_INFORMATION_MBR {
39856|     UCHAR PartitionType;
39857|     BOOLEAN BootIndicator;
39858|     BOOLEAN RecognizedPartition;
39859|     ULONG HiddenSectors;
39860| } PARTITION_INFORMATION_MBR,
39861| | *PPARTITION_INFORMATION_MBR;
39862| //
39863| // The structure SET_PARTITION_INFO_EX is used with the
39864| // ioctl
39865| // IOCTL_SET_PARTITION_INFO_EX to set information about
39866| // a specific
39867| // partition. Note that for MBR partitions, you can
39868| // only set the partition
39869| // signature, whereas GPT partitions allow setting of
39870| // all fields that
39871| // you can get.
39872| //
39873| typedef SET_PARTITION_INFORMATION
39874| | SET_PARTITION_INFORMATION_MBR;
39875| typedef PARTITION_INFORMATION_GPT
39876| | SET_PARTITION_INFORMATION_GPT;
39877| //
39878| typedef struct _SET_PARTITION_INFORMATION_EX {
39879|     PARTITION_STYLE PartitionStyle;
39880|     union {
39881|         SET_PARTITION_INFORMATION_MBR Mbr;
39882|         SET_PARTITION_INFORMATION_GPT Gpt;
39883|     };
39884| } SET_PARTITION_INFORMATION_EX,
39885| | *PSET_PARTITION_INFORMATION_EX;
39886| //
39887| // The structure CREATE_DISK_GPT with the ioctl

```

```

    | IOCTL_DISK_CREATE_DISK
39886| // to initialize an virgin disk with an empty GPT
    | partition table.
39887| //
39888|
39889| typedef struct _CREATE_DISK_GPT {
39890|     GUID DiskId;           // Unique disk id
    | for the disk.
39891|     ULONG MaxPartitionCount; // Maximim number
    | of partitions allowable.
39892| } CREATE_DISK_GPT, *PCREATE_DISK_GPT;
39893|
39894| //
39895| // The structure CREATE_DISK_MBR with the ioctl
    | IOCTL_DISK_CREATE_DISK
39896| // to initialize an virgin disk with an empty MBR
    | partition table.
39897| //
39898|
39899| typedef struct _CREATE_DISK_MBR {
39900|     ULONG Signature;
39901| } CREATE_DISK_MBR, *PCREATE_DISK_MBR;
39902|
39903|
39904| typedef struct _CREATE_DISK {
39905|     PARTITION_STYLE PartitionStyle;
39906|     union {
39907|         CREATE_DISK_MBR Mbr;
39908|         CREATE_DISK_GPT Gpt;
39909|     };
39910| } CREATE_DISK, *PCREATE_DISK;
39911|
39912|
39913| //
39914| // The structure GET_LENGTH_INFORMATION is used with
    | the ioctl
39915| // IOCTL_DISK_GET_LENGTH_INFO to obtain the length, in
    | bytes, of the
39916| // disk, partition, or volume.
39917| //
39918|
39919| typedef struct _GET_LENGTH_INFORMATION {
39920|     LARGE_INTEGER Length;
39921| } GET_LENGTH_INFORMATION, *PGET_LENGTH_INFORMATION;
39922|
39923| //
39924| // The PARTITION_INFORMATION_EX structure is used with
    | the
39925| // IOCTL_DISK_GET_DRIVE_LAYOUT_EX,
    | IOCTL_DISK_SET_DRIVE_LAYOUT_EX,

```



```

39926| // IOCTL_DISK_GET_PARTITION_INFO_EX and
    | IOCTL_DISK_GET_PARTITION_INFO_EX calls.
39927| //
39928|
39929| typedef struct _PARTITION_INFORMATION_EX {
39930|     PARTITION_STYLE PartitionStyle;
39931|     LARGE_INTEGER StartingOffset;
39932|     LARGE_INTEGER PartitionLength;
39933|     ULONG PartitionNumber;
39934|     BOOLEAN RewritePartition;
39935|     union {
39936|         PARTITION_INFORMATION_MBR Mbr;
39937|         PARTITION_INFORMATION_GPT Gpt;
39938|     };
39939| } PARTITION_INFORMATION_EX, *PPARTITION_INFORMATION_EX;
39940|
39941|
39942| //
39943| // GPT specific drive layout information.
39944| //
39945|
39946| typedef struct _DRIVE_LAYOUT_INFORMATION_GPT {
39947|     GUID DiskId;
39948|     LARGE_INTEGER StartingUsableOffset;
39949|     LARGE_INTEGER UsableLength;
39950|     ULONG MaxPartitionCount;
39951| } DRIVE_LAYOUT_INFORMATION_GPT,
    | *PDRIVE_LAYOUT_INFORMATION_GPT;
39952|
39953|
39954| //
39955| // MBR specific drive layout information.
39956| //
39957|
39958| typedef struct _DRIVE_LAYOUT_INFORMATION_MBR {
39959|     ULONG Signature;
39960| } DRIVE_LAYOUT_INFORMATION_MBR,
    | *PDRIVE_LAYOUT_INFORMATION_MBR;
39961|
39962| //
39963| // The structure DRIVE_LAYOUT_INFORMATION_EX is used
    | with the
39964| // IOCTL_SET_DRIVE_LAYOUT_EX and
    | IOCTL_GET_DRIVE_LAYOUT_EX calls.
39965| //
39966|
39967| typedef struct _DRIVE_LAYOUT_INFORMATION_EX {
39968|     ULONG PartitionStyle;
39969|     ULONG PartitionCount;
39970|     union {

```

```

39971|     DRIVE_LAYOUT_INFORMATION_MBR Mbr;
39972|     DRIVE_LAYOUT_INFORMATION_GPT Gpt;
39973| };
39974|     PARTITION_INFORMATION_EX PartitionEntry[1];
39975| } DRIVE_LAYOUT_INFORMATION_EX,
    | *PDRIVE_LAYOUT_INFORMATION_EX;
39976|
39977|
39978| //
39979| // The DISK_GEOMETRY_EX structure is returned on
    | issuing an
39980| // IOCTL_DISK_GET_DRIVE_GEOMETRY_EX ioctl.
39981| //
39982|
39983| typedef enum _DETECTION_TYPE {
39984|     DetectNone,
39985|     DetectInt13,
39986|     DetectExInt13
39987| } DETECTION_TYPE;
39988|
39989| typedef struct _DISK_INT13_INFO {
39990|     USHORT DriveSelect;
39991|     ULONG MaxCylinders;
39992|     USHORT SectorsPerTrack;
39993|     USHORT MaxHeads;
39994|     USHORT NumberDrives;
39995| } DISK_INT13_INFO, *PDISK_INT13_INFO;
39996|
39997| typedef struct _DISK_EX_INT13_INFO {
39998|     USHORT ExBufferSize;
39999|     USHORT ExFlags;
40000|     ULONG ExCylinders;
40001|     ULONG ExHeads;
40002|     ULONG ExSectorsPerTrack;
40003|     ULONG64 ExSectorsPerDrive;
40004|     USHORT ExSectorSize;
40005|     USHORT ExReserved;
40006| } DISK_EX_INT13_INFO, *PDISK_EX_INT13_INFO;
40007|
40008| typedef struct _DISK_DETECTION_INFO {
40009|     ULONG SizeOfDetectInfo;
40010|     DETECTION_TYPE DetectionType;
40011|     union {
40012|         struct {
40013|
40014|             //
40015|             // If DetectionType ==
    | DETECTION_INT13 then we have just the Int13
40016|             // information.
40017|             //

```

```

40018|
40019|         DISK_INT13_INFO Int13;
40020|
40021|         //
40022|         // If DetectionType ==
40023|         | DETECTION_EX_INT13, then we have the
40024|         // extended int 13 information.
40025|         //
40026|         DISK_EX_INT13_INFO ExInt13;
40027|         | // If DetectionType == DetectExInt13
40028|         };
40029|     };
40030| } DISK_DETECTION_INFO, *PDISK_DETECTION_INFO;
40031|
40032| typedef struct _DISK_PARTITION_INFO {
40033|     ULONG SizeOfPartitionInfo;
40034|     PARTITION_STYLE PartitionStyle;
40035|     | // PartitionStyle = RAW, GPT or MBR
40036|     union {
40037|         struct {
40038|             | // If PartitionStyle == MBR
40039|             ULONG Signature;
40040|             | // MBR Signature
40041|             ULONG CheckSum;
40042|             | // MBR CheckSum
40043|             } Mbr;
40044|         struct {
40045|             | // If PartitionStyle == GPT
40046|             GUID DiskId;
40047|             } Gpt;
40048|         };
40049|     } DISK_PARTITION_INFO, *PDISK_PARTITION_INFO;
40050|
40051| //
40052| // The Geometry structure is a variable length
40053| // structure composed of a
40054| // DISK_GEOMETRY_EX structure followed by a
40055| // DISK_PARTITION_INFO structure
40056| // followed by a DISK_DETECTION_DATA structure.
40057| //
40058| #define DiskGeometryGetPartition(Geometry)\
40059|     | ((PDISK_PARTITION_INFO)((Geometry)+1))
40060|
40061| #define DiskGeometryGetDetect(Geometry)\
40062|

```

```

    | ((PDISK_DETECTION_INFO)(((PBYTE)DiskGeometryGetPartition
    | (Geometry)+\
40058|
    | DiskGeometryGetPartition(Geometry)->SizeOfPartitionInfo)
    | ))
40059|
40060| typedef struct _DISK_GEOMETRY_EX {
40061|     DISK_GEOMETRY Geometry;
    | // Standard disk geometry: may be faked by driver.
40062|     LARGE_INTEGER DiskSize;
    | // Must always be correct
40063|     UCHAR Data[1];
    | // Partition, Detect info
40064| } DISK_GEOMETRY_EX, *PDISK_GEOMETRY_EX;
40065|
40066| NTSTATUS PsmCreateFiles( tOpenTransactionInInternal
    | *In, PKEVENT AbortEvent );
40067|
40068|
40069| #endif
40070|
40071|
40072|
40073| File Listing: DEVSUP.cpp
40074|
40075| #include "precomp.h"
40076|
40077| #ifdef ALLOC_PRAGMA_DO_NOT_DO
40078| #pragma alloc_text(PAGE, SbOpenCacheFileInAsyncMode)
40079| #pragma alloc_text(PAGE, SbOpenCacheFileInSyncMode)
40080| #pragma alloc_text(PAGE, SbGetAsyncEvent)
40081| #pragma alloc_text(PAGE, SbReadAndWait)
40082| #pragma alloc_text(PAGE, SbWriteAndWait)
40083| #pragma alloc_text(PAGE, SbGetRegistrySettings)
40084| #pragma alloc_text(PAGE, FailRequest)
40085| #pragma alloc_text(PAGE, FlushVolume)
40086| #endif
40087|
40088| typedef NTSTATUS (*tZwFlushBuffersFile) ( IN HANDLE
    | FileHandle, OUT PIO_STATUS_BLOCK IoStatusBlock );
40089|
40090| tZwFlushBuffersFile ZwFlushBuffersFile;
40091| tZwShutdownSystem ZwShutdownSystem;
40092|
40093|
40094| //-----
    | -----
40095|
40096| #define FSCTL_LOCK_VOLUME
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 6, METHOD_BUFFERED,

```

```

    | FILE_ANY_ACCESS)
40097| #define FSCTL_UNLOCK_VOLUME
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 7, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
40098| #define FSCTL_DISMOUNT_VOLUME
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 8, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
40099| #define FSCTL_INVALIDATE_VOLUMES
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 21, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
40100| #define FSCTL_IS_VOLUME_MOUNTED
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 10, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
40101|
40102|
40103| NTSTATUS FS_SystemCall (
40104|     PFILE_OBJECT   FileObject,
40105|     ULONG           IoControlCode,
40106|     const char      *CallerFunctionName );
40107|
40108|
40109| NTSTATUS FS_DismountVolume( PFILE_OBJECT FileObject )
40110| {
40111|     return FS_SystemCall ( FileObject,
        | FSCTL_DISMOUNT_VOLUME, "FS_DismountVolume" );
40112| }
40113|
40114| NTSTATUS FS_IsVolumeMounted ( PFILE_OBJECT FileObject )
40115| {
40116|     return FS_SystemCall ( FileObject,
        | FSCTL_IS_VOLUME_MOUNTED, "FS_IsVolumeMounted" );
40117| }
40118|
40119|
40120| NTSTATUS FS_UnlockVolume ( PFILE_OBJECT FileObject )
40121| {
40122|     return FS_SystemCall ( FileObject,
        | FSCTL_UNLOCK_VOLUME, "FS_UnlockVolume" );
40123| }
40124|
40125|
40126| typedef NTSTATUS (* FS_SYSTEM_CALL_FUNCTION) (
    | PFILE_OBJECT );
40127|
40128| NTSTATUS Sblo_SystemCall (
40129|     const WCHAR      *VolumeName,
40130|     FS_SYSTEM_CALL_FUNCTION  FsFunctionToCall,
40131|     const char        *ActionDebugText,
40132|     ULONG              DesiredAccess );
40133|

```

```

40134| /*-----
    | -----*/
40135|
40136|
40137| void DeleteAllSnapShots ( void * )
40138| {
40139|     pkSnapshotMaster OneToDelete=NULL;
40140|     PDEVICE_OBJECT DevObj=NULL;
40141|     PFILTERED_EXTENSION DevExt=NULL;
40142|     pkSnapshotEntry p=NULL;
40143|
40144|     Debug(DEBUG_THREAD,("DeleteAllSnapShotsThread:
    | Starting\n"));
40145|
40146|     // start at the top
40147|     __try {
40148|         if(AcquireOpenCloseResource()==STATUS_WAIT_0) {
40149|             __try {
40150|                 UpdateGlobalStatus
    | (PSM_DESTROYING_SNAPSHOT);
40151|                 StartOver:
40152|                     OneToDelete = NULL;
40153|                     DevObj =
    | PSMAN_DRIVER_OBJECT->DeviceObject;
40154|                     // go through all volumes looking for
    | snapshots
40155|                     while(DevObj != NULL) {
40156|
    | if(PsmGetObjectype(DevObj)==OBJECT_FILTEREDDISK) {
40157|                         DevExt =
    | GetFilteredExtension(DevObj);
40158|
40159|                         GetSnapshotForRead();
40160|                         __try {
40161|
    | p=GetTopSnapshot(&DevExt->SnapShots);
40162|                         if(p) {
40163|                             OneToDelete =
    | p->MasterSnapshot;
40164|                             DoneWithSnapshot(p);
40165|                             break;
40166|                         }
40167|                     } __finally {
40168|                         ReleaseSnapshotForRead();
40169|                     }
40170|                 }
40171|                 DevObj=DevObj->NextDevice;
40172|             }
40173|
40174|             if(p) {

```

```

40175|         pOT_USER User =
| FindPSMUser(PsGetCurrentProcess(),(_ETHREAD*)-2);
40176|
| Debug(DEBUG_DCPSM,("DeleteAllSnapShots: Deleting
| Snapshot %08x\n",OneToDelete));
40177|         InternalClosePSM(User,OneToDelete);
40178|         goto StartOver;
40179|     } else {
40180|     }
40181| } __finally {
40182|     ReleaseOpenCloseResource();
40183|     UpdateGlobalStatus (PSM_IDLE);
40184| }
40185| } // if OpenCloseResource acquired
40186| }
| __except(ExceptionFilter(GetExceptionInformation())) {
40187|     Debug(DEBUG_DCPSM,("DeleteAllSnapShots:
| Exception %08x deleting snapshot
| %08x\n",GetExceptionCode(),OneToDelete));
40188| }
40189|
40190| Debug(DEBUG_THREAD,("DeleteAllSnapShotsThread:
| Exiting\n"));
40191| PsTerminateSystemThread( 0 );
40192| return;
40193| }
40194|
40195| void DeleteAllSnapShotsForVolume ( void *Volume )
40196| {
40197|     pkSnapshotMaster OneToDelete=NULL;
40198|     PDEVICE_OBJECT DevObj;
40199|     PFILTERED_EXTENSION DevExt=NULL;
40200|     pkSnapshotEntry p;
40201|
40202|
| Debug(DEBUG_THREAD,("DeleteAllSnapShotsForVolumeThread:
| Starting\n"));
40203|
40204| // start at the top
40205| __try {
40206|     if(AcquireOpenCloseResource()==STATUS_WAIT_0) {
40207|         __try {
40208|             DevObj = (PDEVICE_OBJECT)Volume;
40209|             DevExt = GetFilteredExtension (DevObj);
40210|
40211| StartOver:
40212|             OneToDelete = NULL;
40213|
40214|             GetSnapshotForRead();
40215|         __try {

```

```

40216|
    | p=GetTopSnapShot(&DevExt->SnapShots);
40217|         if(p) {
40218|             OneToDelete =
    | p->MasterSnapShot;
40219|             DoneWithSnapShot(p);
40220|         }
40221|     } __finally {
40222|         ReleaseSnapShotForRead();
40223|     }
40224|
40225|     if(p) {
40226|         pOT_USER User =
    | FindPSMUser(PsGetCurrentProcess(),(_ETHREAD*)-2);
40227|
    | Debug(DEBUG_DCPSM,("DeleteAllSnapShotsForVolume:
    | Deleting Snapshot %08x\n",OneToDelete));
40228|         InternalClosePSM(User,OneToDelete);
40229|         goto StartOver;
40230|     } else {
40231|     }
40232| } __finally {
40233|     ReleaseOpenCloseResource();
40234| }
40235| } // if OpenCloseResource acquired
40236| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
40237|
    | Debug(DEBUG_DCPSM,("DeleteAllSnapShotsForVolume:
    | Exception %08x deleting snapshot
    | %08x\n",GetExceptionCode(),OneToDelete));
40238| }
40239|
40240|
    | Debug(DEBUG_THREAD,("DeleteAllSnapShotsForVolumeThread:
    | Exiting\n"));
40241| PsTerminateSystemThread( 0 );
40242| return;
40243| }
40244|
40245|
40246| /*-----
    | -----*/
40247| void FailRequestAll ( pkSnapShotEntry SnapShot,
    | NTSTATUS Error )
40248| {
40249|     BOOLEAN DoCallbacks=FALSE;
40250|     PAGED_CODE();
40251|
40252|     Debug(DEBUG_DEVSUP,("FailRequest: Error %08x

```



```

    | occurred, failing PSM users\n",Error));
40253|   pmAcquireMutex ( &WorkerThreadMutex, NULL);
40254|   Debug(DEBUG_DEVSUP,("FailRequest: Got mutex\n"));
40255|   // only set error once..
40256|   if(!LastErrorStatus) {
40257|
40258|       Debug(DEBUG_DEVSUP,("FailRequest: Turning psm
    | off\n"));
40259|       LastErrorStatus = Error;
40260|       /* 3-21-99, dont disable drive as it seems that
    | the file system doesnt like it and does
40261|       the following things that are bad mojo.
40262|       1. Lose memory to a vpb as it can not be
    | dismounted right (ie the create fails due to the volume
40263|       being not in the drive when trying to
    | remount, since we can not open the volume, we can not
    | lock
40264|       or dismount it.)
40265|       2. It seems to corrupt memory in the KTHREAD
    | structure for its own thread, not sure why it does this
40266|
40267|       VDiskDisableAll();
40268|       */
40269|       PsmOff();
40270|
40271|       // minimize the amount of time we are in here,
    | as our threads
40272|       // can not do anything, including exit.
40273|       DoCallbacks = TRUE;
40274|   }
40275|
40276|   pmReleaseMutex ( &WorkerThreadMutex );
40277|
40278|   // ok, now call the registered callbacks.
40279|   if(DoCallbacks) {
40280|       pOT_USER User=NULL;
40281|
40282|       Debug(DEBUG_DEVSUP,("FailRequest: Marking
    | snapshots\n"));
40283|       MarkAllSnapShotsWithError( SnapShot, Error );
40284|
40285|       Debug(DEBUG_DEVSUP,("FailRequest: Doing
    | callbacks\n"));
40286|       if((gLogErrors) &&
    | (Error!=PSM_CANCELED_BY_USER) &&
    | (Error!=STATUS_SUCCESS)) {
40287|           WCHAR ErrorStr[10];
40288|           WCHAR *Strings[1];
40289|           swprintf(ErrorStr,L"%08x",Error);
40290|           Strings[0] = ErrorStr;

```

```

40291|
40292|     Debug(DEBUG_DEVSUP,("FailRequest: Logging
    | error %08x\n",Error));
40293|     switch (Error) {
40294|         case PSM_ERROR_CACHEFILE_FULL: {
40295|             /*lint -save -e740 */
40296|
            | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
            | R_CACHEFILE_FULL,PSM_ERROR_CACHEFILE_FULL,NULL,0,NULL,0)
            | ;
40297|             /*lint -restore */
40298|             break;
40299|         }
40300|
40301|     default:
40302|         /*lint -save -e740 */
40303|
            | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_DISA
            | BLED_ERROR,Error,NULL,0,Strings,1);
40304|         /*lint -restore */
40305|     }
40306| }
40307|
40308| if(GlobalData->NumActive) {
40309|     HANDLE TempHandle;
40310|     pmStartThread(
40311|         (PKSTART_ROUTINE)DeleteAllSnapShots,
            | // IN PKSTART_ROUTINE StartRoutine,
40312|         NULL,         // IN PVOID
            | StartContext
40313|         &TempHandle         // OUT
            | PHANDLE ThreadHandle,
40314|     );
40315|     ZwClose(TempHandle);
40316| }
40317|
40318|
40319|     Debug(DEBUG_DEVSUP,("FailRequest: Getting user
        | mutex\n"));
40320|     pmAcquireMutex ( &PSMUserMutex, NULL );
40321|     Debug(DEBUG_DEVSUP,("FailRequest: Got user
        | mutex\n"));
40322|
40323|     User=GlobalData->PSMUsers;
40324|     while(User) {
40325|         if((User->Open) && (User->ErrorEvent)) {
40326|             Debug(DEBUG_DEVSUP,("FailRequest:
                | Setting error event for user %08x\n",User));
40327|             pmSetEvent(User->ErrorEvent);
40328|         } else {

```

```

40329|         Debug(DEBUG_DEVSUP,("FailRequest:
| Skipping user %08x\n",User));
40330|     }
40331|     User=User->Next;
40332| }
40333|     pmReleaseMutex ( &PSMUserMutex);
40334| }
40335|     Debug(DEBUG_DEVSUP,("FailRequest: Leaving
| FailRequest\n"));
40336|
40337| }
40338|
40339| void FailRequestForVolume( pkSnapShotEntry SnapShot,
| NTSTATUS Error )
40340| {
40341|     BOOLEAN DoCallbacks=FALSE;
40342|     PAGED_CODE();
40343|
40344|     Debug(DEBUG_DEVSUP,("FailRequestForVolume: Error
| %08x occured, failing PSM users\n",Error));
40345|     pmAcquireMutex ( &WorkerThreadMutex, NULL);
40346|     Debug(DEBUG_DEVSUP,("FailRequestForVolume: Got
| mutex\n"));
40347|     // only set error once..
40348|     if(!SnapShot->MasterSnapShot->Status) {
40349|         Debug(DEBUG_DEVSUP,("FailRequestForVolume:
| failing\n"));
40350|         SnapShot->MasterSnapShot->Status = Error;
40351|         // minimize the amount of time we are in here,
| as our threads
40352|         // can not do anything, including exit.
40353|         DoCallbacks = TRUE;
40354|     }
40355|
40356|     pmReleaseMutex ( &WorkerThreadMutex );
40357|
40358|     // ok, now call the registered callbacks.
40359|     if(DoCallbacks) {
40360|         pOT_USER User=NULL;
40361|
40362|         Debug(DEBUG_DEVSUP,("FailRequestForVolume:
| Marking snapshots\n"));
40363|         MarkAllSnapShotsWithErrorForVolume( SnapShot,
| Error );
40364|
40365|         Debug(DEBUG_DEVSUP,("FailRequestForVolume:
| Doing callbacks\n"));
40366|         if((gLogErrors) &&
| (Error!=PSM_CANCELED_BY_USER) &&
| (Error!=STATUS_SUCCESS)) {

```

```

40367|         WCHAR ErrorStr[10];
40368|         WCHAR *Strings[1];
40369|         swprintf(ErrorStr,L"%08x",Error);
40370|         Strings[0] = ErrorStr;
40371|
40372|         Debug(DEBUG_DEVSUP,("FailRequestForVolume:
| Logging error %08x\n",Error));
40373|         switch (Error) {
40374|             case PSM_ERROR_CACHEFILE_FULL: {
40375|                 /*lint -save -e740 */
40376|
| LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
| R_CACHEFILE_FULL,PSM_ERROR_CACHEFILE_FULL,NULL,0,NULL,0)
| ;
40377|                 /*lint -restore */
40378|                 break;
40379|             }
40380|
40381|             default:
40382|                 /*lint -save -e740 */
40383|
| LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_DISA
| BLED_ERROR,Error,NULL,0,Strings,1);
40384|                 /*lint -restore */
40385|             }
40386|         }
40387|
40388|         if(GlobalData->NumActive) {
40389|             HANDLE TempHandle;
40390|             pmStartThread(
40391|
| (PKSTART_ROUTINE)DeleteAllSnapShotsForVolume, // IN
| PKSTART_ROUTINE StartRoutine,
40392|             Snapshot->DeviceObject,
| // IN PVOID StartContext
40393|             &TempHandle // OUT
| PHANDLE ThreadHandle,
40394|         );
40395|         ZwClose(TempHandle);
40396|     }
40397|
40398|     // fixfixfix we need to set the error events
| for the snapshots
40399| }
40400|     Debug(DEBUG_DEVSUP,("FailRequestForVolume: Leaving
| FailRequest\n"));
40401|
40402| }
40403|
40404| // decides what action should be taken based on

```

```

    | snapshot and error
40405| void FailRequest( pkSnapShotEntry SnapShot, NTSTATUS
    | Error )
40406| {
40407|     if(SnapShot) {
40408|         switch(Error) {
40409|             case PSM_ERROR_CACHEFILE_FULL :
40410|                 FailRequestForVolume(SnapShot,Error);
40411|                 break;
40412|             default:
40413|                 break;
40414|         }
40415|     } else {
40416|         // global error, not specific to any snapshot
40417|         FailRequestAll(SnapShot,Error);
40418|     }
40419| }
40420|
40421|
40422| char *CopyFileNameOnly( char *Buffer, char *Path )
40423| {
40424|     char *p = strrchr(Path, '\\');
40425|     if( !p ) {
40426|         p = Path;
40427|     }
40428|     if( strlen(p)>12 ) {
40429|         p+=strlen(p)-12;
40430|     }
40431|     strcpy(Buffer,p);
40432|     return Buffer;
40433| }
40434|
40435|
40436| void SbTrace2 ( char *Code, ULONG Arg1, ULONG Arg2,
    | ULONG Arg3, ULONG Arg4, PCHAR Msg, char *File, ULONG
    | Line )
40437| {
40438|     CHAR FileName[20];
40439|
40440|     if(PsmActive) {
40441|         Debug(DEBUG_TRACE,("%-20.20s> %-13.13s %5d:
    | %-08x %-08x %-08x %-08x %s\n",
40442|             Code,
40443|             CopyFileNameOnly(FileName,File),Line,
40444|             Arg1,
40445|             Arg2,
40446|             Arg3,
40447|             Arg4,
40448|             Msg
40449|             ));

```

```

40450| }
40451| }
40452|
40453| #ifdef SYNC
40454| #define SB_DESIRED_ACCESS (FILE_GENERIC_READ |
    | FILE_GENERIC_WRITE)
40455| #else
40456| #define SB_DESIRED_ACCESS (STANDARD_RIGHTS_WRITE |
    | \
40457|          FILE_WRITE_DATA      |
    | \
40458|          FILE_WRITE_ATTRIBUTES |
    | \
40459|          FILE_WRITE_EA        |
    | \
40460|          FILE_APPEND_DATA     |
    | \
40461|          STANDARD_RIGHTS_READ |
    | \
40462|          FILE_READ_DATA       |
    | \
40463|          FILE_READ_ATTRIBUTES |
    | \
40464|          FILE_READ_EA)
40465| #endif
40466|
40467| #define AsyncMode 0
40468| #define SyncMode 1
40469|
40470|
40471| NTSTATUS SbCreateDirectory ( const WCHAR
    | *DirectoryName, PSECURITY_DESCRIPTOR SD, ULONG
    | Attributes )
40472| {
40473|     UNICODE_STRING  UniName={0};
40474|     WCHAR Buffer[200];
40475|     OBJECT_ATTRIBUTES ObjectAttributes={0};
40476|     IO_STATUS_BLOCK IoStatus;
40477|     HANDLE FileHandle;
40478|     NTSTATUS Status;
40479|
40480|     wcscpy(Buffer,DirectoryName);
40481|     if(Buffer[wcslen(Buffer)-1]!=L'\\') {
40482|         wscat(Buffer,L"\\");
40483|     }
40484|     RtlInitUnicodeString(&UniName,Buffer);
40485|
40486|     InitializeObjectAttributes ( &ObjectAttributes,
40487|         &UniName,
40488|         OBJ_CASE_INSENSITIVE,

```

```

40489|             NULL,
40490|             SD );
40491|
40492|     Status = ZwCreateFile( &FileHandle,
40493|             GENERIC_WRITE,
40494|             | // desired access
40495|             &ObjectAttributes,
40496|             | // object attributes
40497|             &IoStatus,
40498|             NULL,          //
40499|             | alloc size
40500|             Attributes,    // file
40501|             | attributes
40502|             FILE_SHARE_WRITE |
40503|             | FILE_SHARE_READ,          // share
40504|             | access
40505|             FILE_CREATE,
40506|             | // create disposition
40507|             FILE_DIRECTORY_FILE|
40508|             0,             // create
40509|             | options
40510|             NULL, // eabuffer
40511|             0 ); // ealength
40512|     if(NT_SUCCESS(Status)) {
40513|         ZwClose(FileHandle);
40514|     }
40515|     return Status;
40516| }
40517|
40518| NTSTATUS SbTouchVolume ( const WCHAR *VolumeName )
40519| {
40520|     UNICODE_STRING  UniName={0};
40521|     WCHAR *Buffer=(WCHAR*)MemAllocateString(256);
40522|     OBJECT_ATTRIBUTES ObjectAttributes={0};
40523|     IO_STATUS_BLOCK IoStatus;
40524|     HANDLE FileHandle;
40525|     NTSTATUS Status;
40526|
40527|     if(!Buffer) {
40528|         return STATUS_INSUFFICIENT_RESOURCES;
40529|     }
40530|
40531|     #ifdef DEBUG
40532|     if(IsSnapShotAcquiredForWrite()) {
40533|         // the reason this is bad is that we have the
40534|         | writer lock
40535|         // and any io that needs to be PSMed, needs to
40536|         | acquire the reader lock
40537|         // thus producing a deadlock, to fix it, dont

```

```

    | call with writer lock
40529|    // which you can do by spinning off whatever
    | you are trying to do to a
40530|    // worker thread.
40531|    Debug(DEBUG_DCPSM,("SbTouchVolume: Snapshot
    | resource acquired for write!\n"));
40532|    DbgBreakPoint();
40533| }
40534| #endif
40535|
40536|    wcscpy(Buffer,VolumeName);
40537|    if(Buffer[wcslen(Buffer)-1]!=L'\\') {
40538|        wcscat(Buffer,L"\\");
40539|    }
40540|    RtlInitUnicodeString(&UniName,Buffer);
40541|
40542|    InitializeObjectAttributes ( &ObjectAttributes,
40543|                                &UniName,
40544|                                OBJ_CASE_INSENSITIVE,
40545|                                NULL,
40546|                                NULL );
40547|
40548|    Status = ZwCreateFile( &FileHandle,
40549|                           GENERIC_READ,
    | // desired access
40550|                           &ObjectAttributes,
    | // object attributes
40551|                           &IoStatus,
40552|                           NULL,          //
    | alloc size
40553|                           FILE_ATTRIBUTE_NORMAL,
    | // file attributes
40554|                           FILE_SHARE_WRITE |
    | FILE_SHARE_READ,          // share
    | access
40555|                           FILE_OPEN,
    | // create disposition
40556|                           FILE_DIRECTORY_FILE|
40557|                           | FILE_OPEN_FOR_BACKUP_INTENT,          // create
    | options
40558|                           NULL, // eabuffer
40559|                           0 ); // ealength
40560|    if(NT_SUCCESS(Status)) {
40561|        ZwClose(FileHandle);
40562|    }
40563|    MemFreeString(Buffer);
40564|    return Status;
40565| }
40566|

```



```

40567| /*-----
| -----*/
40568| NTSTATUS SbGetAsyncEvent (
40569|     HANDLE    &EventHandle,
40570|     PVOID     &EventObject )
40571| {
40572|     NTSTATUS Status=STATUS_SUCCESS;
40573|
40574|     PAGED_CODE();
40575|
40576|     EventHandle = INVALID_HANDLE_VALUE;
40577|     EventObject = NULL;
40578|
40579|     Status = ZwCreateEvent (
40580|         &EventHandle, // OUT PHANDLE EventHandle,
40581|         0,             // IN ACCESS_MASK DesiredAccess,
40582|         NULL,          // IN POBJECT_ATTRIBUTES
| ObjectAttributes,
40583|         SynchronizationEvent, // IN EVENT_TYPE
| EventType,
40584|         FALSE         // IN BOOLEAN InitialEventState
40585|     );
40586|
40587|
40588|     if ( NT_SUCCESS( Status ) ) {
40589|         // Get a Object handle so we can wait on
| requests...
40590|         Status = ObReferenceObjectByHandle(
40591|             EventHandle,    // IN HANDLE Handle,
40592|             SB_DESIRED_ACCESS, // IN ACCESS_MASK
| DesiredAccess,
40593|             NULL,          // IN POBJECT_TYPE
| ObjectType,            // optional
40594|             (KPROCESSOR_MODE)KernelMode,    // IN
| KPROCESSOR_MODE AccessMode,
40595|             &EventObject,    // OUT PVOID *Object,
40596|             NULL             // OUT
| POBJECT_HANDLE_INFORMATION HandleInformation //
| optional
40597|         );
40598|
40599|         if ( NT_SUCCESS( Status ) ) {
40600|             Debug(DEBUG_DEVSUP,("SbGetAsyncEvent:
| EventHandle=%08x,
| EventObject=%08x\n",EventHandle,EventObject));
40601|             ASSERT(IsValidHandle(EventHandle));
40602|             ASSERT(EventObject != NULL);
40603|             return STATUS_SUCCESS;
40604|         } else {
40605|             Debug(DEBUG_DEVSUP,("SbGetAsyncEvent: Error

```

```

    | %08x Unable to get object\n",Status));
40606|     }
40607|     ZwClose(EventHandle);
40608|     EventHandle = INVALID_HANDLE_VALUE;
40609|     EventObject = NULL;
40610| } else {
40611|     Debug(DEBUG_DEVSUP,("SbGetAsyncEvent: Error %08x
    | Unable to create event\n",Status));
40612| }
40613|
40614| // Failure...
40615| ASSERT(!NT_SUCCESS(Status));
40616| ASSERT(EventHandle == INVALID_HANDLE_VALUE);
40617| ASSERT(EventObject == NULL);
40618| return Status;
40619| }
40620|
40621| /*-----
    | -----*/
40622| NTSTATUS SbWriteAndWait (
40623|     pPsmFileInfo Info,
40624|     PIO_STATUS_BLOCK IoStatus,
40625|     PCVOID Buffer,
40626|     ULONG ByteCount,
40627|     PLARGE_INTEGER Location
40628| )
40629| {
40630|     NTSTATUS Status=0;
40631|
40632|     PAGED_CODE();
40633|     ASSERT(GlobalSystemProcessId ==
    | PsGetCurrentProcess());
40634|
40635|     IoStatus->Status = STATUS_PENDING;
40636|     IoStatus->Information = 0xffffffff;
40637|
40638| #ifdef NOWRITE
40639|     if(1) {
40640|         LARGE_INTEGER TimeToWait;
40641|
40642|         Status = 0;
40643|
40644|         TimeToWait.QuadPart =
    | RELATIVE(SECONDS(((Location->QuadPart / 512) &
    | 0xf)+1));
40645|         KeDelayExecutionThread(
40646|             (KPROCESSOR_MODE)KernelMode, // IN
    | KPROCESSOR_MODE WaitMode,
40647|             FALSE, // IN BOOLEAN Alertable,
40648|             &TimeToWait // IN PLARGE_INTEGER Interval

```

```

40649|     );
40650| }
40651| #else
40652| // FILE_WRITE_THROUGH will not return till data is
40653| // written...
40654| ASSERT ( IsValidHandle(Info->WaitHandle) );
40655| ASSERT ( IsValidHandle(Info->WaitObject) );
40656|
40657| Status = ZwWriteFile(
40658|     Info->FileHandle, // IN HANDLE FileHandle,
40659|     Info->WaitHandle, // IN HANDLE Event,
40660|     | optional
40661|     NULL, // IN PIO_APC_ROUTINE
40662|     | ApcRoutine, optional
40663|     NULL, // IN PVOID ApcContext,
40664|     | optional
40665|     IoStatus, // OUT PIO_STATUS_BLOCK
40666|     | IoStatusBlock,
40667|     (PVOID)Buffer, // IN PVOID Buffer,
40668|     ByteCount, // IN ULONG Length,
40669|     Location, // IN PLARGE_INTEGER
40670|     | ByteOffset, optional
40671|     NULL // IN PULONG Key
40672|     | optional
40673| );
40674|
40675| if ( Status != STATUS_SUCCESS ) {
40676|     Debug(DEBUG_DEVSUP,("!!!! SbWriteAndWait:
40677|         | Status=%08x,
40678|         | FileHandle=%08x\n",Status,Info->FileHandle));
40679|     ASSERT(Status!=0xc0000024); // trying to catch
40680|     | bug
40681|     //ASSERT ( Status == STATUS_SUCCESS ); //it's
40682|     | gonna fail, but get attention!
40683| }
40684|
40685| if (Status == STATUS_PENDING) {
40686|     ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
40687|     pmWaitForSingleObject(Info->WaitObject, NULL);
40688|     Status = IoStatus->Status;
40689| }
40690|
40691| #endif
40692|
40693| return Status;
40694| }
40695|
40696| /*-----
40697| | -----*/
40698| NTSTATUS SbReadAndWait (

```

```

40688|         pPsmFileInfo Info,
40689|         PIO_STATUS_BLOCK IoStatus,
40690|         PVOID Buffer,
40691|         ULONG ByteCount,
40692|         PLARGE_INTEGER Location
40693|     )
40694| {
40695|     NTSTATUS Status=0;
40696|
40697|     PAGED_CODE();
40698|     ASSERT(GlobalSystemProcessId ==
        | PsGetCurrentProcess());
40699|
40700|     IoStatus->Status = STATUS_PENDING;
40701|     IoStatus->Information = 0xffffffff;
40702|
40703|     ASSERT ( IsValidHandle(Info->WaitHandle) );
40704|     ASSERT ( IsValidHandle(Info->WaitObject) );
40705|
40706|     Status = ZwReadFile(
40707|         Info->FileHandle, // IN HANDLE FileHandle,
40708|         Info->WaitHandle, // IN HANDLE Event,
        | optional
40709|         NULL, // IN PIO_APC_ROUTINE
        | ApcRoutine, optional
40710|         NULL, // IN PVOID ApcContext,
        | optional
40711|         IoStatus, // OUT PIO_STATUS_BLOCK
        | IoStatusBlock,
40712|         Buffer, // IN PVOID Buffer,
40713|         ByteCount, // IN ULONG Length,
40714|         Location, // IN PLARGE_INTEGER
        | ByteOffset, optional
40715|         NULL // IN PULONG Key
        | optional
40716|     );
40717|
40718|     if ( Status != STATUS_SUCCESS ) {
40719|         Debug(DEBUG_DEVSUP,("!!!! SbReadAndWait:
        | Status=%08x,
        | FileHandle=%08x\n",Status,Info->FileHandle));
40720|         ASSERT(Status!=0xc0000024); // trying to catch
        | bug
40721|         //ASSERT ( Status == STATUS_SUCCESS ); //it's
        | gonna fail, but get attention!
40722|     }
40723|
40724|     if (Status == STATUS_PENDING) {
40725|         ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
40726|         pmWaitForSingleObject(&Info->WaitObject, NULL);

```

```

40727|     Status = IoStatus->Status;
40728| }
40729|
40730| return Status;
40731| }
40732|
40733| /*-----
| -----*/
40734| void SbGetRegistrySettings ( IN PUNICODE_STRING
| RegistryPath )
40735| {
40736|     PAGED_CODE();
40737|
40738|     | Reg_GetULONGKey(RegistryPath,L"FailFreed",1,&gFailFreed)
| ;
40739|
40740|     | Reg_GetULONGKey(RegistryPath,L"LogErrors",1,&gLogErrors)
| ;
40741|
40742|     | Reg_GetULONGKey(RegistryPath,L"LogOpenClose",1,&gLogOpen
| Close);
40743|
40744|     // allow 1-1000 threads
40745|
| Reg_GetULONGKey(RegistryPath,L"NumThreads",NUMTHREADS_DE
| F,&NumberOfThreads);
40746|     if(NumberOfThreads<NUMTHREADS_MIN) {
40747|         NumberOfThreads = NUMTHREADS_MIN;
40748|     }
40749|     if(NumberOfThreads>NUMTHREADS_MAX) {
40750|         NumberOfThreads = NUMTHREADS_MAX;
40751|     }
40752|
40753|     // allow 1-5000000 microseconds (5 seconds)
40754|     // this is number of microseconds in the registry
40755|
| Reg_GetULONGKey(RegistryPath,L"NewThreadStartDelay",NEWT
| HREADDELAY_DEF,&NewThreadStartDelay);
40756|     if(NewThreadStartDelay<NEWTHREADDELAY_MIN) {
40757|         NewThreadStartDelay=NEWTHREADDELAY_MIN;
40758|     }
40759|     if(NewThreadStartDelay>NEWTHREADDELAY_MAX) {
40760|         NewThreadStartDelay=NEWTHREADDELAY_MAX;
40761|     }
40762|
40763|     // allow 1-1000 threads
40764|

```

```

    | Reg_GetULONGKey(RegistryPath,L"MaxThreads",MAXTHREADS_DE
    | F,&MaxThreads);
40765|     if(MaxThreads<NumberOfThreads) {
40766|         MaxThreads = NumberOfThreads;
40767|     }
40768|     if(MaxThreads>MAXTHREADS_MAX) {
40769|         MaxThreads = MAXTHREADS_MAX;
40770|     }
40771|
40772|
    | Reg_GetULONGKey(RegistryPath,L"DoPagingFile",0,&DoPaging
    | File);
40773|     if (DoPagingFile>1) {
40774|         DoPagingFile=1;
40775|     }
40776|
40777|
    | Reg_GetULONGKey(RegistryPath,L"CreateOptions",CREATEOPTI
    | ONS_DEF,&PSManCreateOptions);
40778|
    | Reg_GetULONGKey(RegistryPath,L"OpenOptions",OPENOPTIONS_
    | DEF,&PSManOpenOptions);
40779|
40780|
    | Reg_GetULONGKey(RegistryPath,L"FillOnWrite",FILLONWRITES
    | _DEF,&PSManFillOnWrite);
40781|
40782|     // this is number of microseconds in the registry
40783|
    | Reg_GetULONGKey(RegistryPath,L"HungSystemTimeOut",HUNGSY
    | STEMTIMEOUT_DEF,&gHungSystemTimeOut);
40784|     if(gHungSystemTimeOut<HUNGSYSTEMTIMEOUT_MIN) {
40785|         gHungSystemTimeOut=HUNGSYSTEMTIMEOUT_MIN;
40786|     }
40787|     if(gHungSystemTimeOut>HUNGSYSTEMTIMEOUT_MAX) {
40788|         gHungSystemTimeOut=HUNGSYSTEMTIMEOUT_MAX;
40789|     }
40790|
40791| }
40792|
40793| extern "C" {
40794|     // from ntifs.h
40795|     NTSYSAPI
40796|     BOOLEAN
40797|     NTAPI
40798|     RtlValidSid (
40799|         PSID Sid
40800|     );
40801|     NTSYSAPI
40802|     NTSTATUS

```

```

40803| NTAPI
40804| RtlCreateAcl (
40805|     PACL Acl,
40806|     ULONG AclLength,
40807|     ULONG AclRevision
40808| );
40809| NTSYSAPI
40810| NTSTATUS
40811| NTAPI
40812| RtlAddAccessAllowedAce (
40813|     PACL Acl,
40814|     ULONG AceRevision,
40815|     ACCESS_MASK AccessMask,
40816|     PSID Sid
40817| );
40818|
40819| #define SECURITY_NT_AUTHORITY
| {0,0,0,0,0,5} // ntifs
40820| #define DOMAIN_GROUP_RID_ADMINS
| (0x00000200L)
40821| #define SECURITY_LOCAL_SYSTEM_RID
| (0x00000012L)
40822| #define SECURITY_BUILTIN_DOMAIN_RID
| (0x00000020L)
40823| #define DOMAIN_ALIAS_RID_ADMINS
| (0x00000220L)
40824| }
40825|
40826| /*
40827| ROBLAPTOP\rob is S-1-5-21
40828| NT AUTHORITY\system is S-1-5-18
40829|
40830| The following rights are assigned to the security
| descriptor returned
40831|
40832| System      : Full control
40833| Administrator : Read attributes
40834|              Write attributes
40835|              Delete
40836|              Read permissions
40837|              Change permissions
40838|              Take ownership
40839| */
40840| PSECURITY_DESCRIPTOR SbGetAdminOnlySD( )
40841| {
40842|     NTSTATUS Status;
40843|     PSID adminSid;
40844|     PSID systemSid;
40845|     PACL Acl;
40846|     PSECURITY_DESCRIPTOR SecurityDescriptor;

```

```

40847|    ULONG                systemSidLength;
40848|    SID_IDENTIFIER_AUTHORITY systemSidAuthority =
    | SECURITY_NT_AUTHORITY;
40849|    ULONG                adminSidLength;
40850|    SID_IDENTIFIER_AUTHORITY adminSidAuthority =
    | SECURITY_NT_AUTHORITY;
40851|
40852|    //
40853|    // Initialize buffer pointers
40854|    //
40855|    SecurityDescriptor = (PSECURITY_DESCRIPTOR)
    | MemAllocatePoolWithTag( NonPagedPool, 4096,
    | PSM_SECURITY_TAG );
40856|
40857|    if(SecurityDescriptor) {
40858|        systemSidLength = RtlLengthRequiredSid( 1 );
40859|        adminSidLength = RtlLengthRequiredSid( 2 );
40860|        systemSid = (PSID)
    | (((char*)SecurityDescriptor)+1024);
40861|        adminSid = (PSID)
    | (((char*)systemSid)+systemSidLength);
40862|        Acl = (PACL)(((char*)adminSid)+adminSidLength);
40863|
40864|        //
40865|        // Create an absolute-form security descriptor
    | for manipulation.
40866|        // The one on the security descriptor is in
    | self-relative form.
40867|        //
40868|        Status = RtlCreateSecurityDescriptor(
    | SecurityDescriptor, SECURITY_DESCRIPTOR_REVISION );
40869|
40870|        if( NT_SUCCESS( Status ) ) {
40871|            //
40872|            // Initialize a SID that identifies the
    | world-authority
40873|            //
40874|            RtlInitializeSid( systemSid,
    | &systemSidAuthority, 1 );
40875|            RtlInitializeSid( adminSid,
    | &adminSidAuthority, 2 );
40876|            *RtlSubAuthoritySid( systemSid, 0 ) =
    | SECURITY_LOCAL_SYSTEM_RID;
40877|            *RtlSubAuthoritySid( adminSid, 0 ) =
    | SECURITY_BUILTIN_DOMAIN_RID;
40878|            *RtlSubAuthoritySid( adminSid, 1 ) =
    | DOMAIN_ALIAS_RID_ADMINS;
40879|
40880|
40881|            Status = RtlCreateAcl (

```



```

40882|         Acl, // IN PACL Acl,
40883|         1024, // IN ULONG AclLength,
40884|         ACL_REVISION// IN ULONG AclRevision
40885|     );
40886|
40887|     if(NT_SUCCESS(Status)) {
40888|         Status = RtlAddAccessAllowedAce(
40889|             Acl, // IN OUT PACL Acl,
40890|             ACL_REVISION, // IN ULONG
40891|             | AceRevision,
40892|             SPECIFIC_RIGHTS_ALL |
40893|             | STANDARD_RIGHTS_ALL, // IN ACCESS_MASK AccessMask,
40894|             systemSid// IN PSID Sid
40895|         );
40896|         if(NT_SUCCESS(Status)) {
40897|             Status = RtlAddAccessAllowedAce(
40898|                 Acl, // IN OUT PACL Acl,
40899|                 ACL_REVISION, // IN ULONG
40900|                 | AceRevision,
40901|                 STANDARD_RIGHTS_ALL |
40902|                 STANDARD_RIGHTS_REQUIRED |
40903|                 SYNCHRONIZE |
40904|                 FILE_ADD_FILE | // this is so
40905|                 | we can create files in the directory
40906|                 FILE_READ_ATTRIBUTES |
40907|                 FILE_WRITE_ATTRIBUTES, // IN
40908|                 | ACCESS_MASK AccessMask,
40909|                 adminSid// IN PSID Sid
40910|             );
40911|             Status =
40912|                 | RtlSetDaclSecurityDescriptor(
40913|                     SecurityDescriptor, // IN OUT
40914|                     | PSECURITY_DESCRIPTOR SecurityDescriptor,
40915|                     TRUE, // IN BOOLEAN
40916|                     | DaclPresent,
40917|                     Acl, // IN PACL Dacl
40918|                     | OPTIONAL,
40919|                     FALSE // IN BOOLEAN
40920|                     | DaclDefaulted OPTIONAL
40921|                 );
40922|             if(NT_SUCCESS(Status)) {
40923|                 | Debug(DEBUG_DEVSUP,("SbAssignAdminRightsOnly: Valid
40924|                 | descriptor\n",Status));
40925|                 return SecurityDescriptor;
40926|             } else {
40927|                 | Debug(DEBUG_DEVSUP,("SbAssignAdminRightsOnly: SetDacl
40928|                 | failed %08x\n",Status));

```

```

40918|         }
40919|     } else {
40920|         | Debug(DEBUG_DEVSUP,("SbAssignAdminRightsOnly: AddAce
         | failed %08x\n",Status));
40921|     }
40922|     } else {
40923|         | Debug(DEBUG_DEVSUP,("SbAssignAdminRightsOnly: CreateAcl
         | failed %08x\n",Status));
40924|     }
40925|
40926|     } else {
40927|         | Debug(DEBUG_DEVSUP,("SbAssignAdminRightsOnly: Unable to
         | initialize security descriptor %08x\n",Status));
40928|     }
40929|     MemFreePool(SecurityDescriptor);
40930| } else {
40931|     Status = STATUS_INSUFFICIENT_RESOURCES;
40932|     Debug(DEBUG_DEVSUP,("SbAssignAdminRightsOnly:
         | out of memory for SD %08x\n",Status));
40933| }
40934| return NULL;
40935| }
40936|
40937| #if _WIN32_WINNT<0x0500
40938| #define PsGetVersion(a,b,c,d) (*(c)=1381)
40939| #endif
40940|
40941| /*
40942| It appears under Whistler (build >2250) that
         | ntdll.dll is no longer mapped into the system process
         | space
40943| so we can not longer call the passed in function
         | from a system thread (it would be ok if in the callers
         | context though)
40944|
40945| To work around this, we just take the normal
         | preamble and get the system index number for
         | ZwFlushBuffersFile
40946| (We do this because the index changes with every
         | service pack)
40947|
40948| For every processor architecture we need to have
         | code here to check for it.
40949| This also would happen if Microsoft changed how
         | they call NT functions in future releases
40950| */
40951|

```

```

40952| #if _X86_
40953| // this function is not in the ntoskrnl.lib file. i
    | dont know how to add it, so lets call it direct
40954| //NTSYSAPI NTSTATUS NTAPI ZwFlushBuffersFile ( IN
    | HANDLE FileHandle, OUT PIO_STATUS_BLOCK IoStatusBlock
    | );
40955|
40956| /*-----*/
    | -----*/
40957| /*lint -save -e533 -e144 -e715 */
40958| STATIC ULONG X86_Flush_Index;
40959| STATIC ULONG X86_Shutdown_Index;
40960|
40961| __declspec ( naked ) NTSTATUS NTAPI
    | ZwFlushBuffersFileX86 ( IN HANDLE FileHandle, OUT
    | PIO_STATUS_BLOCK IoStatusBlock )
40962| {
40963|     _asm {
40964|         mov eax,X86_Flush_Index        //;
    | NtFlushBuffersFile call number
40965|         lea edx,dword ptr [esp+4] //; Load effective
    | addr
40966|         int 2Eh                //; System call
40967|         retn 8                  //; Must have ret
    | with naked functions
40968|     }
40969| }
40970|
40971| /*lint -restore */
40972| #endif
40973|
40974| NTSTATUS GrabIndexNumber( PVOID UserModePointer, ULONG
    | &IndexNumber )
40975| {
40976|     // whistler way
40977|     #if _X86_
40978|         unsigned char *Buffer=(unsigned
    | char*)UserModePointer;
40979|         #ifdef DEBUG
40980|             Debug(DEBUG_DEVSUP,("Dump of user mode space\n"));
40981|             Debug(DEBUG_DEVSUP,(" %02x%02x%02x%02x
    | %02x%02x%02x%02x-%02x%02x%02x%02x
    | %02x%02x%02x%02x\n",Buffer[0x00],Buffer[0x01],Buffer[0x0
    | 2],Buffer[0x03],Buffer[0x04],Buffer[0x05],Buffer[0x06],B
    | uffer[0x07],Buffer[0x08],Buffer[0x09],Buffer[0x0a],Buffe
    | r[0x0b],Buffer[0x0c],Buffer[0x0d],Buffer[0x0e],Buffer[0x
    | 0f]));
40982|             Debug(DEBUG_DEVSUP,(" %02x%02x%02x%02x
    | %02x%02x%02x%02x-%02x%02x%02x%02x
    | %02x%02x%02x%02x\n",Buffer[0x10],Buffer[0x11],Buffer[0x1

```

```

| 2],Buffer[0x13],Buffer[0x14],Buffer[0x15],Buffer[0x16],B
| uffer[0x17],Buffer[0x18],Buffer[0x19],Buffer[0x1a],Buffe
| r[0x1b],Buffer[0x1c],Buffer[0x1d],Buffer[0x1e],Buffer[0x
| 1f]));
40983|   Debug(DEBUG_DEVSUP,(" %02x%02x%02x%02x
| %02x%02x%02x%02x-%02x%02x%02x%02x
| %02x%02x%02x%02x\n",Buffer[0x20],Buffer[0x21],Buffer[0x2
| 2],Buffer[0x23],Buffer[0x24],Buffer[0x25],Buffer[0x26],B
| uffer[0x27],Buffer[0x28],Buffer[0x29],Buffer[0x2a],Buffe
| r[0x2b],Buffer[0x2c],Buffer[0x2d],Buffer[0x2e],Buffer[0x
| 2f]));
40984|   Debug(DEBUG_DEVSUP,(" %02x%02x%02x%02x
| %02x%02x%02x%02x-%02x%02x%02x%02x
| %02x%02x%02x%02x\n",Buffer[0x30],Buffer[0x31],Buffer[0x3
| 2],Buffer[0x33],Buffer[0x34],Buffer[0x35],Buffer[0x36],B
| uffer[0x37],Buffer[0x38],Buffer[0x39],Buffer[0x3a],Buffe
| r[0x3b],Buffer[0x3c],Buffer[0x3d],Buffer[0x3e],Buffer[0x
| 3f]));
40985| #endif
40986|
40987|   if((Buffer[0] == 0xb8) && // mov eax,
40988|       (Buffer[5] == 0xba) && // mov edx,
40989|       ((Buffer[10] == 0xff) && (Buffer[11] == 0xd2))
| && // call edx
40990|       ((Buffer[12] == 0xc2) && (Buffer[13] == 0x08))
| // ret 8
40991|   ) {
40992|       // okay, input is what we expect, lets get the
| system index number
40993|       Buffer++;
40994|       IndexNumber = *(ULONG*)Buffer;
40995|       return STATUS_SUCCESS;
40996|   } else {
40997|       // unknown code..
40998|       return STATUS_INVALID_SYSTEM_SERVICE;
40999|   }
41000| #else
41001|   return STATUS_INVALID_SYSTEM_SERVICE;
41002| #endif
41003| }
41004|
41005|
41006| // called in the context of the user
41007| NTSTATUS SetFlushRoutine( PVOID UserModePointer )
41008| {
41009|   ULONG Build=0;
41010|   PsGetVersion( NULL,NULL,&Build,NULL );
41011|   if(Build<=2195) {
41012|       // nt 3.51, 4 and Windows 2000 way
41013|       ZwFlushBuffersFile =

```

```

    | (tZwFlushBuffersFile)UserModePointer;
41014|    } else {
41015|        // whistler way
41016|
    | if(NT_SUCCESS(GrabIndexNumber(UserModePointer,X86_Flush_
    | Index))) {
41017| #ifdef _X86_
41018|        ZwFlushBuffersFile = ZwFlushBuffersFileX86;
41019| #else
41020| #endif
41021|    }
41022| }
41023| return STATUS_SUCCESS;
41024| }
41025|
41026| // called in the context of the user
41027| #if 0
41028| NTSTATUS SetShutdownRoutine( PVOID UserModePointer )
41029| {
41030|     NTSTATUS Status;
41031|     ULONG Build=0;
41032|     PsGetVersion( NULL,NULL,&Build,NULL );
41033|     if(Build<=2195) {
41034|         // nt 3.51, 4 and Windows 2000 way
41035|         ZwShutdownSystem =
            | (tZwShutdownSystem)UserModePointer;
41036|     } else {
41037|         // whistler way
41038|
            | if(NT_SUCCESS(GrabIndexNumber(UserModePointer,X86_Shutdo
            | wn_Index))) {
41039| #ifdef _X86_
41040|         ZwShutdownSystem = ZwShutdownSystemX86;
41041| #else
41042| #endif
41043|     }
41044| }
41045| return STATUS_SUCCESS;
41046| }
41047| #endif
41048|
41049|
41050| NTSTATUS MyFlushBuffersFile( HANDLE Handle,
    | PIO_STATUS_BLOCK IoStatus )
41051| {
41052|     NTSTATUS Status;
41053|
41054|     __try {
41055|         if(ZwFlushBuffersFile) {
41056|             // this causes an write access violation

```

```

    | for some reason..
41057| //
    | ProbeForRead(ZwFlushBuffersFile,4,sizeof(UCHAR));
41058|     Status =
    | ZwFlushBuffersFile(Handle,IoStatus);
41059|     } else {
41060|         Status = STATUS_INVALID_SYSTEM_SERVICE;
41061|     }
41062| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
41063|     Status = GetExceptionCode();
41064|     Debug(DEBUG_DEVSUP,("MyFlushBuffersFile:
    | Exception %08x\n",Status));
41065| }
41066| return Status;
41067| }
41068|
41069| /*-----
    | -----*/
41070| NTSTATUS SbCreateAndFillFile (
41071|     const WCHAR    *FileName,
41072|     PLARGE_INTEGER InitialSize,
41073|     PVOID          AbortEvent,
41074|     BYTE           Pattern,
41075|     ULONG          FillOnWriteOption )
41076| {
41077|     UNICODE_STRING  FullFileName={0};
41078|     OBJECT_ATTRIBUTES ObjectAttributes={0};
41079|     NTSTATUS        Status=0,Status2=0;
41080|     IO_STATUS_BLOCK IoStatus={0};
41081|     ULONG           CreateOptions=0;
41082|     ULONG           DesiredAccess=0;
41083|     HANDLE          FileHandle=NULL;
41084|     FILE_END_OF_FILE_INFORMATION EOFInfo={0};
41085|     FILE_STANDARD_INFORMATION FSInfo={0};
41086|     LARGE_INTEGER Location={0};
41087|     LARGE_INTEGER Timeout={0};
41088|     PSECURITY_DESCRIPTOR SD = SbGetAdminOnlySD();
41089|
41090|     PAGED_CODE();
41091|
41092|     Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
    | FileName='%S'\n",FileName));
41093|     //
    | Reg_GetULONGKey(&gRegistryPath,L"FillOnWrite",FILLONWRIT
    | ES_DEF,&PSManFillOnWrite);
41094|
41095| #ifdef DEBUG
41096|     if(IsSnapshotAcquiredForWrite()) {
41097|         // the reason this is bad is that we have the

```

```

| writer lock
41098| // and any io that needs to be PSMed, needs to
| acquire the reader lock
41099| // thus producing a deadlock, to fix it, dont
| call with writer lock
41100| // which you can do by spinning off whatever
| you are trying to do to a
41101| // worker thread.
41102| Debug(DEBUG_DCPSM,("SbCreateAndFillFile:
| Snapshot resource acquired for write!\n"));
41103| DbgBreakPoint();
41104| }
41105| #endif
41106|
41107| RtlInitUnicodeString( &FullFileName, FileName );
41108|
41109| InitializeObjectAttributes ( &ObjectAttributes,
41110|                               &FullFileName,
41111|                               OBJ_CASE_INSENSITIVE,
41112|                               NULL,
41113|                               SD );
41114|
41115| CreateOptions = FILE_SYNCHRONOUS_IO_NONALERT |
| FILE_WRITE_THROUGH | FILE_SEQUENTIAL_ONLY;
41116| // FILE_NO_COMPRESSION |
41117| // FILE_WRITE_THROUGH |
41118| // FILE_NO_INTERMEDIATE_BUFFERING;
41119|
41120| if(PsManCreateOptions) {
41121| CreateOptions |= PsManCreateOptions;
41122| }
41123|
41124| DesiredAccess = (FILE_GENERIC_READ |
| FILE_GENERIC_WRITE);
41125|
41126| Status = ZwCreateFile( &FileHandle,
41127|                       DesiredAccess,
| // desired access
41128|                       &ObjectAttributes,
| // object attributes
41129|                       &IoStatus,
41130|                       InitialSize,
| // alloc size
41131|                       FILE_ATTRIBUTE_HIDDEN,
| // file attributes
41132|                       0,
| // share access
41133|                       FILE_SUPERSEDE,
| // create disposition
41134|                       CreateOptions,

```

```

    | // create options
41135|         NULL, // eabuffer
41136|         0 ); // ealength
41137|
41138|     EOFInfo.EndOfFile = (*InitialSize);
41139|
41140|     // free security descriptor
41141|     if(SD) {
41142|         MemFreePool(SD);
41143|     }
41144|
41145|     if (NT_SUCCESS(Status)) {
41146|         Status = ZwQueryInformationFile(
41147|             FileHandle,          // IN HANDLE
41148|             | FileHandle,
41149|             &IoStatus,          // OUT
41150|             | PIO_STATUS_BLOCK IoStatusBlock,
41151|             &FSInfo,           // IN PVOID
41152|             | FileInformation,
41153|             sizeof(FILE_STANDARD_INFORMATION),
41154|             | // IN ULONG Length,
41155|             FileStandardInformation// IN
41156|             | FILE_INFORMATION_CLASS FileInformationClass
41157|             );
41158|
41159|         // cant have a directory as a file name
41160|         if(FSInfo.Directory) {
41161|             ZwClose(FileHandle);
41162|             Status = STATUS_FILE_IS_A_DIRECTORY;
41163|             goto EndOfFileStuff;
41164|         }
41165|
41166|         Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
41167|             | AllocSize=%08x%08x\n"
41168|             "After Create
41169|             | EndOfFile=%08x%08x\n"
41170|             "
41171|             | NL=%d, DP=%d, Dir=%d, io=%08x, Inf=%08x\n",
41172|             | FSInfo.AllocationSize.HighPart,
41173|             | FSInfo.AllocationSize.LowPart,
41174|             FSInfo.EndOfFile.HighPart,
41175|             FSInfo.EndOfFile.LowPart,
41176|             FSInfo.NumberOfLinks,
41177|             FSInfo.DeletePending,
41178|             FSInfo.Directory,
41179|             | IoStatus.Status,IoStatus.Information
41180|             ));

```



```

41173|
41174|     Status = ZwSetInformationFile(
41175|         FileHandle,          // IN HANDLE
         | FileHandle,
41176|         &IoStatus,          // OUT
         | PIO_STATUS_BLOCK IoStatusBlock,
41177|         &EOFIInfo,          // IN PVOID
         | FileInformation,
41178|         sizeof(EOFInfo),     // IN ULONG
         | Length,
41179|         FileEndOfFileInformation // IN
         | FILE_INFORMATION_CLASS FileInformationClass
41180|     );
41181|
41182|     if(Status) {
41183|         Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
         | Error %08x setting eof\n",Status));
41184|     }
41185|
41186|     Status = ZwQueryInformationFile(
41187|         FileHandle,          // IN HANDLE
         | FileHandle,
41188|         &IoStatus,          // OUT
         | PIO_STATUS_BLOCK IoStatusBlock,
41189|         &FSInfo,            // IN PVOID
         | FileInformation,
41190|         sizeof(FILE_STANDARD_INFORMATION),
         | // IN ULONG Length,
41191|         FileStandardInformation// IN
         | FILE_INFORMATION_CLASS FileInformationClass
41192|     );
41193|
41194|     Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
         | AllocSize=%08x%08x\n"
41195|         "After Set EOF
         | EndOfFile=%08x%08x\n"
41196|         "
         | NL=%d, DP=%d, Dir=%d\n",
41197|         | FSInfo.AllocationSize.HighPart,
41198|         | FSInfo.AllocationSize.LowPart,
41199|         FSInfo.EndOfFile.HighPart,
41200|         FSInfo.EndOfFile.LowPart,
41201|         FSInfo.NumberOfLinks,
41202|         FSInfo.DeletePending,
41203|         FSInfo.Directory
41204|     ));
41205|
41206|

```

```

41207|     Status = ZwSetInformationFile(
41208|         FileHandle,          // IN HANDLE
41209|         | FileHandle,
41210|         &IoStatus,          // OUT
41211|         | PIO_STATUS_BLOCK IoStatusBlock,
41212|         &EOFIInfo,          // IN PVOID
41213|         | FileInformation,
41214|         sizeof(EOFInfo),     // IN ULONG
41215|         | Length,
41216|         FileAllocationInformation // IN
41217|         | FILE_INFORMATION_CLASS FileInformationClass
41218|     );
41219|
41220|     if(Status) {
41221|         Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
41222|         | Error %08x setting alloc size\n",Status));
41223|     }
41224|
41225|     Status = ZwQueryInformationFile(
41226|         FileHandle,          // IN HANDLE
41227|         | FileHandle,
41228|         &IoStatus,          // OUT
41229|         | PIO_STATUS_BLOCK IoStatusBlock,
41230|         &FSInfo,            // IN PVOID
41231|         | FileInformation,
41232|         sizeof(FILE_STANDARD_INFORMATION),
41233|         | // IN ULONG Length,
41234|         FileStandardInformation// IN
41235|         | FILE_INFORMATION_CLASS FileInformationClass
41236|     );
41237|
41238|     Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
41239|     | AllocSize=%08x%08x\n"
41240|     | "After Set Alloc
41241|     | EndOfFile=%08x%08x\n"
41242|     | "
41243|     | NL=%d, DP=%d, Dir=%d\n",
41244|     | FSInfo.AllocationSize.HighPart,
41245|     | FSInfo.AllocationSize.LowPart,
41246|     | FSInfo.EndOfFile.HighPart,
41247|     | FSInfo.EndOfFile.LowPart,
41248|     | FSInfo.NumberOfLinks,
41249|     | FSInfo.DeletePending,
41250|     | FSInfo.Directory
41251|     ));
41252|
41253|
41254|
41255|
41256|
41257|
41258|
41259|
41260|

```

```

41241|         if ( FillOnWriteOption==FILLONWRITE_DISABLED )
41242|         | {
41243|             // Do nothing.
41244|         } else if(FillOnWriteOption==FILLONWRITE_ALL) {
41245|             ULONG FillSize =
41246|             | min(65536,EOFInfo.EndOfFile.LowPart);
41247|             char *Buffer = (char
41248|             | *)MemAllocatePoolWithTag( PagedPool, FillSize, TEMPTAG
41249|             | );
41250|             if(Buffer) {
41251|                 memset(Buffer,Pattern,FillSize);
41252|                 Location.QuadPart = 0;
41253|                 Status = STATUS_SUCCESS;
41254|                 while(
41255|                     | (Location.QuadPart<EOFInfo.EndOfFile.QuadPart) &&
41256|                     | (Status == STATUS_SUCCESS)){
41257| Retry:
41258|                     Status = ZwWriteFile(
41259|                     | FileHandle, // IN HANDLE
41260|                     | FileHandle,
41261|                     | NULL, // IN HANDLE Event,
41262|                     | optional
41263|                     | NULL, // IN
41264|                     | PIO_APC_ROUTINE ApcRoutine, optional
41265|                     | NULL, // IN PVOID
41266|                     | ApcContext, optional
41267|                     | &IoStatus, // OUT
41268|                     | PIO_STATUS_BLOCK IoStatusBlock,
41269|                     | Buffer, // IN PVOID
41270|                     | Buffer,
41271|                     | FillSize, // IN ULONG
41272|                     | Length,
41273|                     | &Location, // IN
41274|                     | PLARGE_INTEGER ByteOffset, optional
41275|                     | NULL // IN PULONG
41276|                     | Key optional
41277|                     );
41278|             | if(Status==STATUS_INSUFFICIENT_RESOURCES) {
41279|                 LARGE_INTEGER TimeToWait={0};
41280|                 Debug(DEBUG_DCPSM,("SbCreateAndFillFile: Out of memory
41281|                 | from ZwWriteFile, delaying\n"));
41282|                 TimeToWait.QuadPart =
41283|                 | RELATIVE(SECONDS(1));
41284|                 KeDelayExecutionThread(

```

```

41273|
| (KPROCESSOR_MODE)KernelMode, // IN KPROCESSOR_MODE
| WaitMode,
41274|                                     FALSE,
| // IN BOOLEAN Alertable,
41275|
| &TimeToWait // IN PLARGE_INTEGER Interval
41276|                                     );
41277|
41278|
41279|         goto Retry;
41280|     }
41281|     Location.QuadPart += FillSize;
41282|
41283|     if(NT_SUCCESS(Status) &&
| (AbortEvent)) {
41284|         // see if abort event signaled,
| 0 timeout checks
41285|         Status =
| pmWaitForSingleObject(AbortEvent,&Timeout);
41286|         if(Status==STATUS_SUCCESS) {
41287|             // event to cancel was
| specified!
41288|             Status =
| PSM_CANCELED_BY_USER;
41289|         } else {
41290|             Status = STATUS_SUCCESS;
41291|         }
41292|     }
41293| }
41294|
41295|     FREE_POINTER(Buffer);
41296|     Debug(DEBUG_DEVSUP,("Done writing to
| '%S', status=%08x\n",FileName,Status));
41297|     Status2 = ZwQueryInformationFile(
41298|         FileHandle,           // IN
| HANDLE FileHandle,
41299|         &IoStatus,           // OUT
| PIO_STATUS_BLOCK IoStatusBlock,
41300|         &FSInfo,             // IN
| PVOID FileInformation,
41301|         sizeof(FILE_STANDARD_INFORMATION),
| // IN ULONG Length,
41302|         FileStandardInformation// IN
| FILE_INFORMATION_CLASS FileInformationClass
41303|     );
41304|
41305|     Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
| AllocSize=%08x%08x\n"

```

```

41306|                                     "After fill
    | EndOfFile=%08x%08x\n"
41307|                                     "
    | NL=%d, DP=%d, Dir=%d\n",
41308|
    | FSInfo.AllocationSize.HighPart,
41309|
    | FSInfo.AllocationSize.LowPart,
41310|
    | FSInfo.EndOfFile.HighPart,
41311|
    | FSInfo.EndOfFile.LowPart,
41312|
    | FSInfo.NumberOfLinks,
41313|
    | FSInfo.DeletePending,
41314|                                     FSInfo.Directory
41315|                                     ));
41316|
41317|     } else {
41318|         Debug(DEBUG_DEVSUP,("Out of memory for
    | writing to '%S'\n",FileName));
41319|         ASSERT(FALSE);
41320|     }
41321| } else if(FillOnWriteOption==FILLONWRITE_EOF) {
41322|     CHAR Buffer=0;
41323|     // write to end of file
41324|     Location.QuadPart =
    | EOFInfo.EndOfFile.QuadPart - 1;
41325|
41326|     Status = ZwWriteFile(
41327|         FileHandle,      // IN HANDLE
    | FileHandle,
41328|         NULL,            // IN HANDLE
    | Event,                [optional]
41329|         NULL,            // IN
    | PIO_APC_ROUTINE ApcRoutine,    [optional]
41330|         NULL,            // IN PVOID
    | ApcContext,          [optional]
41331|         &IoStatus,       // OUT
    | PIO_STATUS_BLOCK IoStatusBlock,
41332|         &Buffer,         // IN PVOID
    | Buffer,
41333|         sizeof(Buffer),  // IN ULONG
    | Length,
41334|         &Location,       // IN
    | PLARGE_INTEGER ByteOffset,    [optional]
41335|         NULL              // IN PULONG Key
    | [optional]
41336|     );

```

```

41337|     } else {
41338|         Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
| Undefined
| FillOnWriteOption=%08x\n",FillOnWriteOption));
41339|         ASSERT(FALSE);
41340|     }
41341|
41342|     // flush the data, NTFS seems to keep this
| stuff around and not actually "extend" the file
41343|     // in sp4
41344|     Status2 =
| MyFlushBuffersFile(FileHandle,&IoStatus);
41345|
41346|     if(!NT_SUCCESS(Status2)) {
41347|         Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
| Error %08x (%08x) flushing
| '%S'\n",Status2,IoStatus.Status,FileName));
41348|     } else {
41349|         Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
| success flushing '%S'\n",FileName));
41350|     }
41351|
41352|     Status2 = ZwQueryInformationFile(
41353|         FileHandle,           // IN HANDLE
| FileHandle,
41354|         &IoStatus,           // OUT
| PIO_STATUS_BLOCK IoStatusBlock,
41355|         &FSInfo,             // IN PVOID
| FileInformation,
41356|         sizeof(FILE_STANDARD_INFORMATION),
| // IN ULONG Length,
41357|         FileStandardInformation// IN
| FILE_INFORMATION_CLASS FileInformationClass
41358|     );
41359|
41360|     Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
| AllocSize=%08x%08x\n"
41361|         "After flush
| EndOfFile=%08x%08x\n"
41362|         "
| NL=%d, DP=%d, Dir=%d\n",
41363|         FSInfo.AllocationSize.HighPart,
41364|         FSInfo.AllocationSize.LowPart,
41365|         FSInfo.EndOfFile.HighPart,
41366|         FSInfo.EndOfFile.LowPart,
41367|         FSInfo.NumberOfLinks,
41368|         FSInfo.DeletePending,
41369|         FSInfo.Directory

```

```

41370|         ));
41371|
41372|
41373|         if(!NT_SUCCESS(Status)) {
41374|             FILE_DISPOSITION_INFORMATION Del={0};
41375|
41376|             // delete file if we could not create it.
            | gets around having 0 length files
41377|             Del.DeleteFile = TRUE;
41378|             Status2 = ZwSetInformationFile( FileHandle,
41379|                 &IoStatus, &Del, sizeof(Del),
41380|                 FileDispositionInformation
41381|             );
41382|             if(NT_SUCCESS(Status2)) {
41383|                 | Debug(DEBUG_DEVSUP,("SbCreateAndFillFile: Success
            | deleting file\n"));
41384|             } else {
41385|                 | Debug(DEBUG_DEVSUP,("SbCreateAndFillFile: Error %08x
            | (%08x) deleting file\n",Status2,IoStatus.Status));
41386|             }
41387|         }
41388|
41389|         // keep the file closed,
41390|         ZwClose(FileHandle);
41391|         FileHandle=NULL;
41392|     }
41393|
41394| EndOfFileStuff:
41395|     if(NT_SUCCESS(Status)) {
41396|         Debug(DEBUG_DEVSUP,("SbCreateAndFillFile: '%S'
            | created %08x
            | (%08x)\n",FileName,Status,IoStatus.Status));
41397|     } else {
41398|         Debug(DEBUG_DEVSUP,("SbCreateAndFillFile:
            | Error! Create '%S' returned = %08x
            | (%08x)\n",FileName,Status,IoStatus.Status));
41399|     }
41400|     return Status;
41401| }
41402|
41403|
41404|
41405| /*-----
            | -----*/
41406| NTSTATUS SblsADirectory ( const WCHAR *CacheFileName )
41407| {
41408|     UNICODE_STRING FullFileName={0};
41409|     OBJECT_ATTRIBUTES ObjectAttributes={0};

```

```

41410| NTSTATUS      Status=0;
41411| IO_STATUS_BLOCK IoStatus={0};
41412| HANDLE          FileHandle=NULL;
41413| FILE_STANDARD_INFORMATION FSInfo={0};
41414|
41415| PAGED_CODE();
41416|
41417| RtlInitUnicodeString( &FullFileName,
    | CacheFileName);
41418|
41419| InitializeObjectAttributes ( &ObjectAttributes,
41420|                             &FullFileName,
41421|                             OBJ_CASE_INSENSITIVE,
41422|                             NULL,
41423|                             NULL );
41424|
41425| Status = ZwCreateFile( &FileHandle,
41426|                       FILE_GENERIC_READ,
    | // desired access
41427|                       &ObjectAttributes,
    | // object attributes
41428|                       &IoStatus,
41429|                       NULL,          //
    | alloc size
41430|                       FILE_ATTRIBUTE_NORMAL,
    | // file attributes
41431|                       FILE_SHARE_WRITE |
    | FILE_SHARE_READ,          // share
    | access
41432|                       FILE_OPEN,
    | // create disposition
41433|                       | FILE_SYNCHRONOUS_IO_NONALERT,          // create
    | options
41434|                       NULL, // eabuffer
41435|                       0 ); // ealength
41436|
41437| if (NT_SUCCESS(Status)) {
41438|     Status = ZwQueryInformationFile(
41439|         FileHandle,          // IN HANDLE
    | FileHandle,
41440|         &IoStatus,          // OUT
    | PIO_STATUS_BLOCK IoStatusBlock,
41441|         &FSInfo,           // IN PVOID
    | FileInformation,
41442|         sizeof(FILE_STANDARD_INFORMATION),
    | // IN ULONG Length,
41443|         FileStandardInformation// IN
    | FILE_INFORMATION_CLASS FileInformationClass
41444|     );

```



```

41445|
41446|     if(FSInfo.Directory) {
41447|         Status = STATUS_FILE_IS_A_DIRECTORY;
41448|     }
41449|
41450|     // keep the file closed, as this routine is to
        | create the cache file
41451|     // not open it..
41452|     ZwClose(FileHandle);
41453|     FileHandle=NULL;
41454| }
41455|
41456| if((Status!=STATUS_FILE_IS_A_DIRECTORY) && (Status
        | != STATUS_OBJECT_PATH_SYNTAX_BAD)) {
41457|     Debug(DEBUG_DEVSUP,("SbIsADirectory: Cache file
        | is not a directory %08x\n",Status));
41458| } else {
41459|     Debug(DEBUG_DEVSUP,("SbIsADirectory: Error!
        | cache file is a directory %08x\n",Status));
41460| }
41461| return Status;
41462| }
41463|
41464| /*-----
        | -----*/
41465| NTSTATUS SbDeleteFile(
41466|     const WCHAR *FileName,
41467|     ULONG FileAttributes )
41468| {
41469|     UNICODE_STRING FullFileName={0};
41470|     OBJECT_ATTRIBUTES ObjectAttributes={0};
41471|     NTSTATUS Status=0;
41472|     IO_STATUS_BLOCK IoStatus={0};
41473|     HANDLE FileHandle=NULL;
41474|
41475|     PAGED_CODE();
41476|
41477| #ifdef DEBUG
41478|     if(IsSnapshotAcquiredForWrite()) {
41479|         // the reason this is bad is that we have the
            | writer lock
41480|         // and any io that needs to be PSMed, needs to
            | acquire the reader lock
41481|         // thus producing a deadlock, to fix it, dont
            | call with writer lock
41482|         // which you can do by spinning off whatever
            | you are trying to do to a
41483|         // worker thread.
41484|         Debug(DEBUG_DCPSM,("SbDeleteFile: Snapshot
            | resource acquired for write!\n"));

```

```

41485|     DbgBreakPoint();
41486| }
41487| #endif
41488|
41489| RtlInitUnicodeString( &FullFileName, FileName);
41490|
41491| InitializeObjectAttributes ( &ObjectAttributes,
41492|                             &FullFileName,
41493|                             OBJ_CASE_INSENSITIVE,
41494|                             NULL,
41495|                             NULL );
41496|
41497| Status = ZwCreateFile( &FileHandle,
41498|                       FILE_GENERIC_READ |
41499| | FILE_GENERIC_WRITE,      // desired access
41500|                       &ObjectAttributes,
41501| | // object attributes
41502|                       &IoStatus,
41503|                       NULL,      //
41504| | alloc size
41505|                       FileAttributes,    //
41506| | file attributes
41507|                       0,
41508| | // share access
41509|                       FILE_OPEN,
41510| | // create disposition
41511| | FILE_SYNCHRONOUS_IO_NONALERT,      // create
41512| | options
41513|                       NULL, // eabuffer
41514|                       0 ); // ealength
41515|
41516| if (NT_SUCCESS(Status)) {
41517|     FILE_DISPOSITION_INFORMATION Del={0};
41518|
41519|     // delete file if we could not create it. gets
41520|     | around having 0 length files
41521|     Del.DeleteFile = TRUE;
41522|     Status = ZwSetInformationFile( FileHandle,
41523|                                   &IoStatus, &Del, sizeof(Del),
41524|                                   FileDispositionInformation
41525| );
41526|
41527|     ZwClose(FileHandle);
41528|
41529|     if ( !NT_SUCCESS(Status) ) {
41530|         Debug(DEBUG_DEVSUP,("SbDeleteFile:
41531| | ZwSetInformationFile('%S') returned %08x\n",Status));
41532|     }
41533| } else {

```

```

41525|     Debug(DEBUG_DEVSUP,("SbDeleteFile:
    | ZwCreateFile('%S') returned %08x\n",FileName,Status));
41526| }
41527|
41528| if(NT_SUCCESS(Status)) {
41529|     Debug(DEBUG_DEVSUP,("SbDeleteFile: Success
    | deleting file\n"));
41530| } else {
41531|     Debug(DEBUG_DEVSUP,("SbDeleteFile: Error %08x
    | (%08x) deleting file
    | '%S'\n",Status,IoStatus.Status,FileName));
41532| }
41533| return Status;
41534| }
41535|
41536|
41537| extern "C"
41538| NTSYSAPI
41539| NTSTATUS
41540| NTAPI
41541| ZwWaitForSingleObject(
41542|     IN HANDLE Handle,
41543|     IN BOOLEAN Alertable,
41544|     IN PLARGE_INTEGER Timeout OPTIONAL
41545| );
41546|
41547| //-----
    | -----
41548|
41549|
41550| // NOTE. We needed to do the following nesting delink
    | in a separate thread because it eats up too much stack
41551| //     when it's being performed during rebuild on a
    | reboot.
41552|
41553|
41554| typedef struct sDeleteReparseInterface {
41555|     IN const WCHAR *FileName;
41556|     OUT NTSTATUS Status;
41557| } tDeleteReparseInterface, *pDeleteReparseInterface;
41558| /*-----
    | -----*/
41559| void SbDeleteAllReparsePointsAndDirThread(
    | tDeleteReparseInterface *IF )
41560| {
41561|     UNICODE_STRING FullFileName={0};
41562|     OBJECT_ATTRIBUTES ObjectAttributes={0};
41563| // NTSTATUS Status=0;
41564| IO_STATUS_BLOCK IoStatus={0};
41565| HANDLE FileHandle=NULL;

```

```

41566|    WCHAR NewName[512];
41567|
41568|    PAGED_CODE();
41569|
41570|    #if DO_ALL_CLEANUP
41571|
41572|        | Debug(DEBUG_DEVSUP,("SbDeleteAllReparsePointsAndDirThrea
41573|        | d: path='%S'\n",IF->FileName));
41574|    #endif /*DO_ALL_CLEANUP*/
41575|
41576|    RtlInitUnicodeString( &FullFileName, IF->FileName);
41577|
41578|    InitializeObjectAttributes ( &ObjectAttributes,
41579|                                &FullFileName,
41580|                                OBJ_CASE_INSENSITIVE,
41581|                                NULL,
41582|                                NULL );
41583|
41584|    #if 1 //what it was
41585|        IF->Status = ZwCreateFile( &FileHandle,
41586|                                FILE_TRAVERSE |
41587|                                FILE_DELETE_CHILD |
41588|                                FILE_READ_ATTRIBUTES |
41589|                                FILE_LIST_DIRECTORY,
41590|                                // desired access
41591|                                &ObjectAttributes,
41592|                                // object attributes
41593|                                &IoStatus,
41594|                                NULL,          //
41595|                                // alloc size
41596|                                FILE_ATTRIBUTE_NORMAL,
41597|                                // file attributes
41598|                                0,
41599|                                // share access
41600|                                FILE_OPEN,
41601|                                // create disposition
41602|                                FILE_DIRECTORY_FILE,
41603|                                NULL, // eabuffer
41604|                                0 ); // ealength
41605|    #else //what we tried different
41606|        IF->Status = ZwCreateFile( &FileHandle,
41607|                                FILE_ALL_ACCESS,
41608|                                // desired access
41609|                                &ObjectAttributes,
41610|                                // object attributes
41611|                                &IoStatus,
41612|                                NULL,          //
41613|                                // alloc size
41614|                                FILE_ATTRIBUTE_NORMAL,
41615|                                // file attributes
41616|                                0,
41617|                                // share access
41618|                                FILE_OPEN,
41619|                                // create disposition
41620|                                FILE_DIRECTORY_FILE,
41621|                                NULL, // eabuffer
41622|                                0 ); // ealength
41623|    #endif
41624|
41625|    // If we failed to create the file, we need to delete the file
41626|    // and then try to create it again.
41627|    if (IF->Status != STATUS_SUCCESS)
41628|    {
41629|        // If we failed to create the file, we need to delete the file
41630|        // and then try to create it again.
41631|        IF->Status = ZwDeleteFile( &FileHandle,
41632|                                FILE_DELETE_CHILD,
41633|                                // desired access
41634|                                &ObjectAttributes,
41635|                                // object attributes
41636|                                &IoStatus,
41637|                                NULL,          //
41638|                                // alloc size
41639|                                FILE_ATTRIBUTE_NORMAL,
41640|                                // file attributes
41641|                                0,
41642|                                // share access
41643|                                FILE_OPEN,
41644|                                // create disposition
41645|                                FILE_DIRECTORY_FILE,
41646|                                NULL, // eabuffer
41647|                                0 ); // ealength
41648|    }
41649|
41650|    // If we failed to create the file, we need to delete the file
41651|    // and then try to create it again.
41652|    if (IF->Status != STATUS_SUCCESS)
41653|    {
41654|        // If we failed to create the file, we need to delete the file
41655|        // and then try to create it again.
41656|        IF->Status = ZwDeleteFile( &FileHandle,
41657|                                FILE_DELETE_CHILD,
41658|                                // desired access
41659|                                &ObjectAttributes,
41660|                                // object attributes
41661|                                &IoStatus,
41662|                                NULL,          //
41663|                                // alloc size
41664|                                FILE_ATTRIBUTE_NORMAL,
41665|                                // file attributes
41666|                                0,
41667|                                // share access
41668|                                FILE_OPEN,
41669|                                // create disposition
41670|                                FILE_DIRECTORY_FILE,
41671|                                NULL, // eabuffer
41672|                                0 ); // ealength
41673|    }
41674|
41675|    // If we failed to create the file, we need to delete the file
41676|    // and then try to create it again.
41677|    if (IF->Status != STATUS_SUCCESS)
41678|    {
41679|        // If we failed to create the file, we need to delete the file
41680|        // and then try to create it again.
41681|        IF->Status = ZwDeleteFile( &FileHandle,
41682|                                FILE_DELETE_CHILD,
41683|                                // desired access
41684|                                &ObjectAttributes,
41685|                                // object attributes
41686|                                &IoStatus,
41687|                                NULL,          //
41688|                                // alloc size
41689|                                FILE_ATTRIBUTE_NORMAL,
41690|                                // file attributes
41691|                                0,
41692|                                // share access
41693|                                FILE_OPEN,
41694|                                // create disposition
41695|                                FILE_DIRECTORY_FILE,
41696|                                NULL, // eabuffer
41697|                                0 ); // ealength
41698|    }
41699|
41700|    // If we failed to create the file, we need to delete the file
41701|    // and then try to create it again.
41702|    if (IF->Status != STATUS_SUCCESS)
41703|    {
41704|        // If we failed to create the file, we need to delete the file
41705|        // and then try to create it again.
41706|        IF->Status = ZwDeleteFile( &FileHandle,
41707|                                FILE_DELETE_CHILD,
41708|                                // desired access
41709|                                &ObjectAttributes,
41710|                                // object attributes
41711|                                &IoStatus,
41712|                                NULL,          //
41713|                                // alloc size
41714|                                FILE_ATTRIBUTE_NORMAL,
41715|                                // file attributes
41716|                                0,
41717|                                // share access
41718|                                FILE_OPEN,
41719|                                // create disposition
41720|                                FILE_DIRECTORY_FILE,
41721|                                NULL, // eabuffer
41722|                                0 ); // ealength
41723|    }
41724|
41725|    // If we failed to create the file, we need to delete the file
41726|    // and then try to create it again.
41727|    if (IF->Status != STATUS_SUCCESS)
41728|    {
41729|        // If we failed to create the file, we need to delete the file
41730|        // and then try to create it again.
41731|        IF->Status = ZwDeleteFile( &FileHandle,
41732|                                FILE_DELETE_CHILD,
41733|                                // desired access
41734|                                &ObjectAttributes,
41735|                                // object attributes
41736|                                &IoStatus,
41737|                                NULL,          //
41738|                                // alloc size
41739|                                FILE_ATTRIBUTE_NORMAL,
41740|                                // file attributes
41741|                                0,
41742|                                // share access
41743|                                FILE_OPEN,
41744|                                // create disposition
41745|                                FILE_DIRECTORY_FILE,
41746|                                NULL, // eabuffer
41747|                                0 ); // ealength
41748|    }
41749|
41750|    // If we failed to create the file, we need to delete the file
41751|    // and then try to create it again.
41752|    if (IF->Status != STATUS_SUCCESS)
41753|    {
41754|        // If we failed to create the file, we need to delete the file
41755|        // and then try to create it again.
41756|        IF->Status = ZwDeleteFile( &FileHandle,
41757|                                FILE_DELETE_CHILD,
41758|                                // desired access
41759|                                &ObjectAttributes,
41760|                                // object attributes
41761|                                &IoStatus,
41762|                                NULL,          //
41763|                                // alloc size
41764|                                FILE_ATTRIBUTE_NORMAL,
41765|                                // file attributes
41766|                                0,
41767|                                // share access
41768|                                FILE_OPEN,
41769|                                // create disposition
41770|                                FILE_DIRECTORY_FILE,
41771|                                NULL, // eabuffer
41772|                                0 ); // ealength
41773|    }
41774|
41775|    // If we failed to create the file, we need to delete the file
41776|    // and then try to create it again.
41777|    if (IF->Status != STATUS_SUCCESS)
41778|    {
41779|        // If we failed to create the file, we need to delete the file
41780|        // and then try to create it again.
41781|        IF->Status = ZwDeleteFile( &FileHandle,
41782|                                FILE_DELETE_CHILD,
41783|                                // desired access
41784|                                &ObjectAttributes,
41785|                                // object attributes
41786|                                &IoStatus,
41787|                                NULL,          //
41788|                                // alloc size
41789|                                FILE_ATTRIBUTE_NORMAL,
41790|                                // file attributes
41791|                                0,
41792|                                // share access
41793|                                FILE_OPEN,
41794|                                // create disposition
41795|                                FILE_DIRECTORY_FILE,
41796|                                NULL, // eabuffer
41797|                                0 ); // ealength
41798|    }
41799|
41800|    // If we failed to create the file, we need to delete the file
41801|    // and then try to create it again.
41802|    if (IF->Status != STATUS_SUCCESS)
41803|    {
41804|        // If we failed to create the file, we need to delete the file
41805|        // and then try to create it again.
41806|        IF->Status = ZwDeleteFile( &FileHandle,
41807|                                FILE_DELETE_CHILD,
41808|                                // desired access
41809|                                &ObjectAttributes,
41810|                                // object attributes
41811|                                &IoStatus,
41812|                                NULL,          //
41813|                                // alloc size
41814|                                FILE_ATTRIBUTE_NORMAL,
41815|                                // file attributes
41816|                                0,
41817|                                // share access
41818|                                FILE_OPEN,
41819|                                // create disposition
41820|                                FILE_DIRECTORY_FILE,
41821|                                NULL, // eabuffer
41822|                                0 ); // ealength
41823|    }
41824|
41825|    // If we failed to create the file, we need to delete the file
41826|    // and then try to create it again.
41827|    if (IF->Status != STATUS_SUCCESS)
41828|    {
41829|        // If we failed to create the file, we need to delete the file
41830|        // and then try to create it again.
41831|        IF->Status = ZwDeleteFile( &FileHandle,
41832|                                FILE_DELETE_CHILD,
41833|                                // desired access
41834|                                &ObjectAttributes,
41835|                                // object attributes
41836|                                &IoStatus,
41837|                                NULL,          //
41838|                                // alloc size
41839|                                FILE_ATTRIBUTE_NORMAL,
41840|                                // file attributes
41841|                                0,
41842|                                // share access
41843|                                FILE_OPEN,
41844|                                // create disposition
41845|                                FILE_DIRECTORY_FILE,
41846|                                NULL, // eabuffer
41847|                                0 ); // ealength
41848|    }
41849|
41850|    // If we failed to create the file, we need to delete the file
41851|    // and then try to create it again.
41852|    if (IF->Status != STATUS_SUCCESS)
41853|    {
41854|        // If we failed to create the file, we need to delete the file
41855|        // and then try to create it again.
41856|        IF->Status = ZwDeleteFile( &FileHandle,
41857|                                FILE_DELETE_CHILD,
41858|                                // desired access
41859|                                &ObjectAttributes,
41860|                                // object attributes
41861|                                &IoStatus,
41862|                                NULL,          //
41863|                                // alloc size
41864|                                FILE_ATTRIBUTE_NORMAL,
41865|                                // file attributes
41866|                                0,
41867|                                // share access
41
```

```

    | FILE_SHARE_READ|FILE_SHARE_WRITE|FILE_SHARE_DELETE,
    | // share access
41604|          FILE_OPEN,
    | // create disposition
41605|
    | FILE_DIRECTORY_FILE|FILE_SYNCHRONOUS_IO_NONALERT,
41606|          NULL, // eabuffer
41607|          0 ); // ealength
41608| #endif
41609|  if (NT_SUCCESS(IF->Status)) {
41610|      FILE_DIRECTORY_INFORMATION *Fi=NULL;
41611|      ULONG NumFound=0;
41612|      ULONG NumSuccessfullyDeleted=0;
41613|      ULONG NumUnsuccessfullyDeleted=0;
41614|      HANDLE Event;
41615|
41616|      ZwCreateEvent (
41617|          &Event, // OUT PHANDLE EventHandle,
41618|          EVENT_ALL_ACCESS,          // IN
    | ACCESS_MASK DesiredAccess,
41619|          NULL,          // IN POBJECT_ATTRIBUTES
    | ObjectAttributes,
41620|          SynchronizationEvent, // IN EVENT_TYPE
    | EventType,
41621|          FALSE          // IN BOOLEAN
    | InitialEventState
41622|      );
41623|
41624|
41625|      ULONG FiSize =
    | sizeof(FILE_DIRECTORY_INFORMATION)+sizeof(WCHAR)*256;
41626|      Fi =
    | (FILE_DIRECTORY_INFORMATION*)MemAllocatePoolWithTag(Page
    | dPool,FiSize,TEMPTAG);
41627|      if(Fi) {
41628|          do {
41629|              IF->Status = ZwQueryDirectoryFile(
41630|                  FileHandle,
41631|                  Event,
41632|                  NULL,
41633|                  NULL,
41634|                  &IoStatus,
41635|                  Fi,
41636|                  FiSize,
41637|                  FileDirectoryInformation,
41638|                  TRUE,
41639|                  NULL,
41640|                  FALSE
41641|              );
41642|              if(IF->Status==STATUS_PENDING) {

```

```

41643|          ASSERT(KeGetCurrentIrql() <
| DISPATCH_LEVEL);
41644|          | ZwWaitForSingleObject(Event,FALSE,NULL);
41645|
41646|          IF->Status = IoStatus.Status;
41647|          }
41648|
41649|          if(NT_SUCCESS(IF->Status)) {
41650|              NTSTATUS Status2=0;
41651|              NumFound++;
41652|              ULONG CharsInFileName =
| Fi->FileNameLength / sizeof(WCHAR);
41653|          | swprintf(NewName,L"%-*s",CharsInFileName,CharsInFileNa
| me,Fi->FileName);
41654|          if ( wcscmp(NewName,L".")!=0 &&
| wcscmp(NewName,L"..")!=0 ) {
41655|              // Only delete reparse points
| so we dont accidentally delete data files.
41656|              // Also delete empty
| directories because they might be failed junctions.
41657|              if(Fi->FileAttributes &
| (FILE_ATTRIBUTE_REPARSE_POINT |
| FILE_ATTRIBUTE_DIRECTORY)) {
41658|          | swprintf(NewName,L"%s\\%-*s",IF->FileName,CharsInFileN
| ame,CharsInFileName,Fi->FileName);
41659|
41660|              if((Status2=
| SbDeleteMountPoint(NewName))==0) {
41661|                  #if DO_ALL_CLEANUP
41662|          | Debug(DEBUG_DEVSUP,("SbDeleteAllReparsePointsAndDirThrea
| d: Deleted file
| '%-*S'\n",Fi->FileNameLength/sizeof(WCHAR),Fi->FileNam
| eLength/sizeof(WCHAR),Fi->FileName));
41663|                  #endif
| /*DO_ALL_CLEANUP*/
41664|          | NumSuccessfullyDeleted++;
41665|              } else {
41666|                  #if DO_ALL_CLEANUP
41667|          | Debug(DEBUG_DEVSUP,("SbDeleteAllReparsePointsAndDirThrea
| d: Error %08x deleting file
| '%-*S'\n",Status2,Fi->FileNameLength/sizeof(WCHAR),Fi-
| >FileNameLength/sizeof(WCHAR),Fi->FileName));
41668|                  #endif
| /*DO_ALL_CLEANUP*/

```

```

41669|
    | NumUnsuccessfullyDeleted++;
41670|    }
41671|    } else {
41672|        #if DO_ALL_CLEANUP
41673|
    | Debug(DEBUG_DEVSUP,("SbDeleteAllReparsePointsAndDirThrea
    | d: Skipping file '%S'\n",NewName));
41674|        #endif /*DO_ALL_CLEANUP*/
41675|    }
41676|    } else {
41677|        #if DO_ALL_CLEANUP
41678|
    | Debug(DEBUG_DEVSUP,("SbDeleteAllReparsePointsAndDirThrea
    | d: Skipping special dir '%S'\n",NewName));
41679|        #endif /*DO_ALL_CLEANUP*/
41680|    }
41681|    }
41682|    } while(IF->Status==0);
41683|
41684|    MemFreePool(Fi);
41685|    }
41686|
41687|    ZwClose(Event);
41688|    ZwClose(FileHandle);
41689|
41690|    #if DO_ALL_CLEANUP
41691|
    | Debug(DEBUG_DEVSUP,("SbDeleteAllReparsePointsAndDirThrea
    | d results (Status=%08x):\n",IF->Status));
41692|    Debug(DEBUG_DEVSUP,(" Total files found
    | : %d\n",NumFound));
41693|    Debug(DEBUG_DEVSUP,(" Number deleted
    | : %d\n",NumSuccessfullyDeleted));
41694|    Debug(DEBUG_DEVSUP,(" Number not deleted
    | : %d\n",NumUnsuccessfullyDeleted));
41695|    #endif /*DO_ALL_CLEANUP*/
41696|    }
41697|
41698| #ifdef DEBUG
41699|    #if DO_ALL_CLEANUP
41700|        if(NT_SUCCESS(IF->Status)) {
41701|
    | Debug(DEBUG_DEVSUP,("SbDeleteAllReparsePointsAndDirThrea
    | d: Success deleting file\n"));
41702|        } else {
41703|
    | Debug(DEBUG_DEVSUP,("SbDeleteAllReparsePointsAndDirThrea
    | d: Error %08x (%08x)\n",IF->Status,loStatus.Status));
41704|        }

```

```

41705|  #endif /*DO_ALL_CLEANUP*/
41706| #endif /*DEBUG*/
41707| }
41708|
41709| /*-----
    | -----*/
41710| void SbSnapShotCleanupThread ( tDeleteReparseInterface
    | *IF )
41711| {
41712|  // typical IF->FileName =
    | '\Device\PsmDevices_0200\_Device_HarddiskVolume3_1\snapshots'
41713|
41714|  // This function is used only for cleaning up a
    | virtual volume.
41715|  // It calls SbDeleteAllReparsePointsAndDirThread to
    | clean up
41716|  // the same stuff we do on the real volume, plus
    | anything else
41717|  // we additionally want to clean up only on
    | snapshots.
41718|
41719|  Debug(DEBUG_DEVSUP,("SbSnapShotCleanupThread:
    | path='%S'\n",IF->FileName));
41720|  SbDeleteAllReparsePointsAndDirThread (IF);
41721|
41722|  // Remove pagefile.sys from virtual volume.
41723|  // That's OK because even if we revert to snapshot,
    | pagefile.sys will
41724|  // be re-created.
41725|
41726|  ULONG DirNameLengthInChars      =
    | wcslen(IF->FileName);
41727|  const ULONG PathLengthInChars   = 256 +
    | DirNameLengthInChars;
41728|  const ULONG PathLengthInBytes   =
    | sizeof(WCHAR) * PathLengthInChars;
41729|  WCHAR *PathName = (WCHAR
    | *)MemAllocatePoolWithTag(PagedPool, PathLengthInBytes,
    | FILENAMETAG);
41730|  if ( PathName ) {
41731|      wcscpy ( PathName, IF->FileName );
41732|      if ( DirNameLengthInChars>0 &&
    | PathName[DirNameLengthInChars-1]=='\\' ) {
41733|          PathName[--DirNameLengthInChars] = 0; //
    | remove final backslash
41734|      }
41735|
41736|      // Scan backwards to final remaining
    | backslash... truncate afterward

```



```

41737|      // to eliminate 'snapshots' directory. (We
      | want root of virtual volume.)
41738|      while ( DirNameLengthInChars>0 &&
      | PathName[DirNameLengthInChars-1]!='\\' ) {
41739|          --DirNameLengthInChars;
41740|      }
41741|
41742|      if ( DirNameLengthInChars > 0 ) {
41743|          PathName[DirNameLengthInChars] = 0;
41744|
      | Debug(DEBUG_DEVSUP,("SbSnapShotCleanupThread:
      | Extracted virtual volume root = \"%S\\n\",PathName));
41745|
41746|          struct FileDeleteInfo {
41747|              const WCHAR *      FileName;
41748|              ULONG              FileAttributes;
41749|          };
41750|
41751|          static FileDeleteInfo ExtraFilesToDelete[]
      | = {
41752|              {L"pagefile.sys",
      | (FILE_ATTRIBUTE_HIDDEN | FILE_ATTRIBUTE_SYSTEM)},
41753|              {NULL,0} // marks end of list
41754|          };
41755|
41756|          for ( ULONG i=0;
      | ExtraFilesToDelete[i].FileName != NULL; ++i ) {
41757|              wcsncpy (
      | &PathName[DirNameLengthInChars],
      | ExtraFilesToDelete[i].FileName );
41758|              SbDeleteFile (PathName,
      | ExtraFilesToDelete[i].FileAttributes);
41759|          }
41760|      }
41761|
41762|      FREE_POINTER(PathName);
41763|  } else {
41764|      Debug(DEBUG_DEVSUP,("SbSnapShotCleanupThread:
      | Out of memory for path name!\\n"));
41765|  }
41766| }
41767|
41768|
41769| /*-----*/
      | -----*/
41770| NTSTATUS SbDeleteAllReparsePointsAndDir( const WCHAR
      | *FileName )
41771| {
41772|     tDeleteReparseInterface IF;
41773|     HANDLE TempHandle;

```

```

41774| NTSTATUS StartThreadStatus = 0;
41775|
41776| IF.FileName = FileName;
41777| IF.Status = 0;
41778|
41779| #ifdef DEBUG
41780| if(IsSnapShotAcquiredForWrite()) {
41781|     // the reason this is bad is that we have the
    | writer lock
41782|     // and any io that needs to be PSMed, needs to
    | acquire the reader lock
41783|     // thus producing a deadlock, to fix it, dont
    | call with writer lock
41784|     // which you can do by spinning off whatever
    | you are trying to do to a
41785|     // worker thread.
41786|
    | Debug(DEBUG_DCPSM,("SbDeleteAllReparsePointsAndDir:
    | Snapshot resource acquired for write!\n"));
41787|     DbgBreakPoint();
41788| }
41789| #endif
41790|
41791| #if DO_ALL_CLEANUP
41792| Debug(DEBUG_DEVSUP,("PSMAN:
    | SbDeleteAllReparsePointsAndDirThread, starting work
    | thread!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    | !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n"));
41793| #endif /*DO_ALL_CLEANUP*/
41794|
41795| StartThreadStatus = pmStartThread(
41796|     | (PKSTART_ROUTINE)SbDeleteAllReparsePointsAndDirThread,
    | // IN PKSTART_ROUTINE StartRoutine,
41797|     (PVOID)&IF, // IN
    | PVOID StartContext
41798|     &TempHandle // OUT
    | PHANDLE ThreadHandle,
41799|     );
41800| if(NT_SUCCESS(StartThreadStatus)) {
41801|     #if DO_ALL_CLEANUP
41802|     Debug(DEBUG_DEVSUP,("PSMAN: System ready,
    | SbDeleteAllReparsePointsAndDirThread started,
    | waiting\n"));
41803|     #endif /*DO_ALL_CLEANUP*/
41804|
41805|     ZwWaitForSingleObject(TempHandle,FALSE,NULL);
41806|
41807|     // dont need the handle anymore.
41808|     ZwClose(TempHandle);

```

```

41809|     TempHandle=NULL;
41810|
41811|     // the thread sets the irp status
41812| } else {
41813|     Debug(DEBUG_DEVSUP,("Error %08x starting
| SbDeleteAllReparsePointsAndDirThread\n",StartThreadStatu
| s));
41814|     IF.Status = StartThreadStatus;
41815| }
41816| return (IF.Status);
41817| }
41818|
41819| /*-----
| -----*/
41820| NTSTATUS SbSnapShotCleanup ( const WCHAR *FileName )
41821| {
41822|     tDeleteReparseInterface IF;
41823|     HANDLE TempHandle;
41824|     NTSTATUS StartThreadStatus = 0;
41825|
41826|     IF.FileName = FileName;
41827|     IF.Status = 0;
41828|
41829|
41830|     Debug(DEBUG_DEVSUP,("PSMAN: SbSnapShotCleanup,
| starting work
| thread!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
| !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n"));
41831|     StartThreadStatus = pmStartThread(
41832|         | (PKSTART_ROUTINE)SbSnapShotCleanupThread, // IN
| PKSTART_ROUTINE StartRoutine,
41833|         (PVOID)&IF, // IN
| PVOID StartContext
41834|         &TempHandle // OUT
| PHANDLE ThreadHandle,
41835|     );
41836|     if(NT_SUCCESS(StartThreadStatus)) {
41837|         Debug(DEBUG_DEVSUP,("PSMAN: System ready,
| SbSnapShotCleanup started, waiting\n"));
41838|         ZwWaitForSingleObject(TempHandle,FALSE,NULL);
41839|
41840|         // dont need the handle anymore.
41841|         ZwClose(TempHandle);
41842|         TempHandle=NULL;
41843|
41844|         // the thread sets the irp status
41845|     } else {
41846|         Debug(DEBUG_DEVSUP,("Error %08x starting
| SbSnapShotCleanupThread\n",StartThreadStatus));

```

```

41847|     IF.Status = StartThreadStatus;
41848| }
41849| return (IF.Status);
41850| }
41851|
41852| /*-----
| -----*/
41853| NTSTATUS SbDeleteMountPoint( const WCHAR *FileName )
41854| {
41855|     UNICODE_STRING  FullFileName={0};
41856|     OBJECT_ATTRIBUTES ObjectAttributes={0};
41857|     NTSTATUS        Status=0;
41858|     IO_STATUS_BLOCK IoStatus={0};
41859|     HANDLE          FileHandle=NULL;
41860|
41861|     PAGED_CODE();
41862|
41863|     RtlInitUnicodeString( &FullFileName, FileName);
41864|
41865|     InitializeObjectAttributes ( &ObjectAttributes,
41866|                                  &FullFileName,
41867|                                  OBJ_CASE_INSENSITIVE,
41868|                                  NULL,
41869|                                  NULL );
41870|
41871|     Status = ZwCreateFile( &FileHandle,
41872|                           FILE_GENERIC_WRITE |
41873|                           | FILE_GENERIC_READ, // desired access
41874|                           &ObjectAttributes,
41875|                           | // object attributes
41876|                           &IoStatus,
41877|                           NULL,
41878|                           | // alloc size
41879|                           FILE_ATTRIBUTE_NORMAL,
41880|                           | // file attributes
41881|                           0, // share access
41882|                           FILE_OPEN,
41883|                           | // create disposition
41884|                           FILE_DIRECTORY_FILE|
41885|                           | FILE_OPEN_REPARSE_POINT|
41886|                           | FILE_OPEN_FOR_BACKUP_INTENT, // create
41887|                           | options
41888|                           NULL, // eabuffer
41889|                           0 ); // ealength
41890|
41891|     if (NT_SUCCESS(Status)) {
41892|         FILE_DISPOSITION_INFORMATION Del={0};
41893|

```

```

41888|    // delete file if we could not create it.  gets
      | around having 0 length files
41889|    Del.DeleteFile = TRUE;
41890|    Status = ZwSetInformationFile( FileHandle,
41891|        &IoStatus, &Del, sizeof(Del),
41892|        FileDispositionInformation
41893|    );
41894|
41895|    ZwClose(FileHandle);
41896|
41897|    if ( !NT_SUCCESS(Status) ) {
41898|        Debug(DEBUG_DEVSUP,("SbDeleteMountPoint:
      | ZwSetInformationFile returned %08x\n",Status));
41899|    }
41900| }
41901|
41902| if(NT_SUCCESS(Status)) {
41903|     #if DO_ALL_CLEANUP
41904|         Debug(DEBUG_DEVSUP,("SbDeleteMountPoint:
      | Success deleting mount point '%S'\n",FileName));
41905|         #endif /*DO_ALL_CLEANUP*/
41906|     } else {
41907|         Debug(DEBUG_DEVSUP,("SbDeleteMountPoint: Error
      | %08x (%08x) deleting mount point
      | '%S'\n",Status,IoStatus.Status,FileName));
41908|     }
41909|     return Status;
41910| }
41911|
41912|
41913| /*-----
      | -----*/
41914|
41915| NTSTATUS SbIo_WriteDevice (
41916|     PDEVICE_OBJECT DeviceObject,
41917|     PLARGE_INTEGER ByteOffset,
41918|     ULONG          ByteCount,
41919|     const char      *Buff )
41920| {
41921|     PIRP              Irp=NULL;
41922|     KEVENT             Event={0};
41923|     IO_STATUS_BLOCK    IoStatusBlock={0};
41924|     NTSTATUS           Status=0;
41925|
41926|     PAGED_CODE();
41927|
41928|     __try {
41929|         //
41930|         // Set the event object to the unsigaled
      | state.

```

```

41931|    // It will be used to signal request
    | completion.
41932|    //
41933|
41934|    KeInitializeEvent(&Event,
41935|                    NotificationEvent,
41936|                    FALSE);
41937|
41938|
41939|
41940|    //
41941|    // Create IRP for read
41942|    //
41943|
41944|    Irp = IoBuildSynchronousFsdRequest(
    | IRP_MJ_WRITE,
41945|    | DeviceObject,
41946|                                (PVOID)
    | Buff,
41947|                                ByteCount,
41948|                                ByteOffset,
41949|                                &Event,
41950|    | &IoStatusBlock );
41951|
41952|    if (!Irp) {
41953|        Debug(DEBUG_DEVSUP,("Sblo_WriteDevice:
    | Error! Unable to allocate irp\n"));
41954|        return STATUS_INSUFFICIENT_RESOURCES;
41955|    }
41956|
41957|    Status = IoCallDriver(DeviceObject, Irp);
41958|
41959|    if (Status == STATUS_PENDING) {
41960|
41961|        ASSERT(KeGetCurrentIrql() <
    | DISPATCH_LEVEL);
41962|        pmWaitForSingleObject(&Event, NULL);
41963|
41964|        Status = IoStatusBlock.Status;
41965|    }
41966| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
41967|     Status = GetExceptionCode();
41968|     Debug(DEBUG_DEVSUP,("Sblo_WriteDevice:
    | Exception %08x\n",Status));
41969| }
41970|
41971| return Status;

```

```

41972| }
41973|
41974| /*-----
| -----*/
41975| NTSTATUS Sblo_ReadDevice( PDEVICE_OBJECT DeviceObject,
41976|                           PLARGE_INTEGER ByteOffset,
41977|                           ULONG      ByteCount,
41978|                           char      *Buff)
41979| {
41980|     PIRP          Irp=NULL;
41981|     KEVENT         Event={0};
41982|     IO_STATUS_BLOCK IoStatusBlock={0};
41983|     NTSTATUS       Status=0;
41984|
41985|     PAGED_CODE();
41986|
41987|     __try {
41988| /*
41989|         Debug(DEBUG_DEVSUP,("Sblo_ReadDevice: Reading
| Device %08x at %08x%08x for %08x to %08x\n",
41990|         DeviceObject,
41991|         ByteOffset->HighPart,
41992|         ByteOffset->LowPart,
41993|         ByteCount,
41994|         Buff
41995|         ));
41996| */
41997|         //
41998|         // Set the event object to the unsigned
| state.
41999|         // It will be used to signal request
| completion.
42000|         //
42001|
42002|         //Debug(DEBUG_DEVSUP,("Setting Event\n"));
42003|         KeInitializeEvent(&Event,
42004|             NotificationEvent,
42005|             FALSE);
42006|
42007|
42008|
42009|         //
42010|         // Create IRP for read
42011|         //
42012|
42013|         //Debug(DEBUG_DEVSUP,("Creating Read Irp\n"));
42014|         Irp = IoBuildSynchronousFsdRequest(
| IRP_MJ_READ,
42015|         DeviceObject,

```

```

42016|          Buff,
42017|          ByteCount,
42018|          ByteOffset,
42019|          &Event,
42020|
42021|      | &IoStatusBlock);
42022|
42022|      if (!Irp) {
42023|          Debug(DEBUG_DEVSUP,("Sblo_ReadDevice:
42024|      | Error! Unable to allocate irp\n"));
42025|          return STATUS_INSUFFICIENT_RESOURCES;
42026|      }
42027|
42027|      Status = IoCallDriver(DeviceObject, Irp);
42028|
42029|      if (Status == STATUS_PENDING) {
42030|
42031|      //          Debug(DEBUG_DEVSUP,("Sblo_ReadDevice:
42032|      | Waiting for read to finish Irp=%08x, Event=%08x,
42033|      | IoStatus=%08x\n",Irp,Event,&IoStatusBlock));
42034|          ASSERT(KeGetCurrentIrql() <
42035|      | DISPATCH_LEVEL);
42036|          pmWaitForSingleObject(&Event, NULL);
42037|
42038|          Status = IoStatusBlock.Status;
42039|      }
42040|      }
42041|
42042|      __except(ExceptionFilter(GetExceptionInformation())) {
42043|          Status = GetExceptionCode();
42044|          Debug(DEBUG_DEVSUP,("Sblo_ReadDevice: Exception
42045|      | %08x\n",Status));
42046|      }
42047|
42048|      return Status;
42049| }
42050|
42051| /*-----*/
42052| | -----*/
42053| NTSTATUS
42054| SbIo_ReadDeviceMdlCompletionRoutine(
42055|     IN PDEVICE_OBJECT DeviceObject,
42056|     IN PIRP          Irp,
42057|     IN PVOID          Context
42058| )
42059| {
42060|     NOT_REFERENCED(DeviceObject);
42061|     NOT_REFERENCED(Context);
42062|
42063|     pmSetEvent(Irp->UserEvent);
42064| }

```



```

42058|    // keep nt from touching our irp, which will be
      | handled by person
42059|    // who wanted the event set
42060|    return STATUS_MORE_PROCESSING_REQUIRED;
42061| }
42062|
42063| /*-----*/
      | -----*/
42064| NTSTATUS Sblo_ReadDeviceMdl( PDEVICE_OBJECT
      | DeviceObject,
42065|                             PLARGE_INTEGER ByteOffset,
42066|                             ULONG          ByteCount,
42067|                             PIRP
      | OriginalIrp,
42068|                             PMDL          Mdl)
42069| {
42070|     PIRP          Irp=NULL;
42071|     KEVENT        Event={0};
42072|     IO_STATUS_BLOCK IoStatusBlock={0};
42073|     NTSTATUS       Status=0;
42074|     PIO_STACK_LOCATION Stack=NULL;
42075|
42076|     PAGED_CODE();
42077|
42078|     __try {
42079| /*
42080|         Debug(DEBUG_DEVSUP,("Sblo_ReadDevice: Reading
      | Device %08x at %08x%08x for %08x to %08x\n",
42081|         DeviceObject,
42082|         ByteOffset->HighPart,
42083|         ByteOffset->LowPart,
42084|         ByteCount,
42085|         Buff
42086|         ));
42087| */
42088|         //
42089|         // Set the event object to the unsigned
      | state.
42090|         // It will be used to signal request
      | completion.
42091|         //
42092|         KeInitializeEvent(&Event, NotificationEvent,
      | FALSE);
42093|
42094|
42095|         //
42096|         // Create IRP for read
42097|         //
42098|
42099|         Irp = IrpAllocateIrp(DeviceObject->StackSize);

```

```

42100|
42101|     if (!Irp) {
42102|         Debug(DEBUG_DEVSUP,("Sblo_ReadDeviceMdl:
| Error! Unable to allocate irp\n"));
42103|         return STATUS_INSUFFICIENT_RESOURCES;
42104|     }
42105|
42106|     Irp->Flags = 0; //IRP_READ_OPERATION;
42107|
42108| /* TESTTEST
42109|     if(OriginalIrp->Flags & IRP_NOCACHE) {
42110|         Irp->Flags |= IRP_NOCACHE;
42111|     }
42112|     if(OriginalIrp->Flags & IRP_PAGING_IO) {
42113|         Irp->Flags |= IRP_PAGING_IO;
42114|     }
42115|     if(OriginalIrp->Flags &
| IRP_SYNCHRONOUS_PAGING_IO) {
42116|         Irp->Flags |= IRP_SYNCHRONOUS_PAGING_IO;
42117|     }
42118| */
42119|     Irp->AssociatedIrp.SystemBuffer = NULL;
42120|     Irp->MdlAddress = Mdl;
42121|     Irp->UserBuffer = NULL;
42122|     Irp->UserEvent = &Event;
42123|     Irp->UserIoSb = &IoStatusBlock;
42124|     /*lint -save -e740 */
42125|     Irp->Tail.Overlay.Thread =
| PsGetCurrentThread();
42126|     /*lint -restore */
42127|     if(OriginalIrp) {
42128|         Irp->Tail.Overlay.OriginalFileObject =
| OriginalIrp->Tail.Overlay.OriginalFileObject;
42129|     }
42130|     Irp->RequestorMode =
| (KPROCESSOR_MODE)KernelMode;
42131|     Irp->IoStatus.Status =
| STATUS_PENDING;
42132|     Irp->IoStatus.Information = 0;
42133|
42134|     Stack = IoGetNextIrpStackLocation(Irp);
42135|     RtlZeroMemory((PVOID)Stack,
| sizeof(IO_STACK_LOCATION));
42136|     IoSetCompletionRoutine(Irp,
| Sblo_ReadDeviceMdlCompletionRoutine, NULL, TRUE, TRUE,
| TRUE);
42137|     Stack->MajorFunction = IRP_MJ_READ;
42138|     Stack->MinorFunction = 0;
42139|     Stack->Parameters.Read.ByteOffset.QuadPart =
| ByteOffset->QuadPart;

```

```

42140|     Stack->Parameters.Read.Length = ByteCount;
42141|     Stack->Parameters.Read.Key = 0;
42142|     Stack->DeviceObject = DeviceObject;
42143|     Stack->FileObject = NULL;
42144|
42145|     Status = IoCallDriver(DeviceObject, Irp);
42146|
42147|     if (Status == STATUS_PENDING) {
42148|
42149| //         Debug(DEBUG_DEVSUP,("Sblo_ReadDevice:
| Waiting for read to finish Irp=%08x, Event=%08x,
| Status=%08x,
| IoStatus=%08x,%08x\n",Irp,&Event,Status,IoStatusBlock.St
| atus,IoStatusBlock.Information));
42150|         ASSERT(KeGetCurrentIrql() <
| DISPATCH_LEVEL);
42151|         pmWaitForSingleObject(&Event,NULL);
42152|
42153|         Status = IoStatusBlock.Status;
42154|     } else {
42155|         Debug(DEBUG_DEVSUP,("Sblo_ReadDevice: Read
| finished without wait. Irp=%08x, Event=%08x,
| Status=%08x,
| IoStatus=%08x,%08x\n",Irp,&Event,Status,IoStatusBlock.St
| atus,IoStatusBlock.Information));
42156|     }
42157|
42158|     // free the Irp we allocated, as we do not need
| it anymore.
42159|     IrpFreeIrp(Irp);
42160|
42161| }
| __except(ExceptionFilter(GetExceptionInformation())) {
42162|     Status = GetExceptionCode();
42163|     Debug(DEBUG_DEVSUP,("Sblo_ReadDeviceMdl:
| Exception %08x\n",Status));
42164| }
42165|
42166| return Status;
42167| }
42168|
42169|
42170| NTSTATUS Sblo_OpenVolumeHandle (
42171|     const WCHAR    *VolumeName,
42172|     HANDLE          &VolumeHandle,
42173|     PFILE_OBJECT    &VolumeFileObject,
42174|     ULONG           DesiredAccess )
42175| {
42176|     UNICODE_STRING  UniName={0};
42177|     OBJECT_ATTRIBUTES ObjectAttributes={0};

```

```

42178| IO_STATUS_BLOCK    IoStatus = {0};
42179| HANDLE              FileHandle = 0;
42180| NTSTATUS             Status = 0;
42181|
42182| VolumeHandle = INVALID_HANDLE_VALUE;
42183| VolumeFileObject = NULL;
42184|
42185| RtlInitUnicodeString(&UniName,VolumeName);
42186|
42187| InitializeObjectAttributes (
42188|     &ObjectAttributes,
42189|     &UniName,
42190|     OBJ_CASE_INSENSITIVE,
42191|     NULL,
42192|     NULL );
42193|
42194| Status = ZwCreateFile(
42195|     &VolumeHandle,
42196|     DesiredAccess,
42197|     &ObjectAttributes,
42198|     &IoStatus,
42199|     NULL,                                //
    | alloc size
42200|     FILE_ATTRIBUTE_NORMAL,
42201|     FILE_SHARE_WRITE | FILE_SHARE_READ,
42202|     FILE_OPEN,
42203|     0,                                    //
    | create options
42204|     NULL,                                //
    | eabuffer
42205|     0 );                                //
    | ealength
42206|
42207| if(NT_SUCCESS(Status)) {
42208|     Status = ObReferenceObjectByHandle(
42209|         VolumeHandle,
42210|         GENERIC_WRITE,
42211|         NULL,
42212|         (KPROCESSOR_MODE)KernelMode,
42213|         (PVOID *)&VolumeFileObject,
42214|         NULL );
42215|
42216|     if(!NT_SUCCESS(Status)) {
42217|         Debug(DEBUG_DEVSUP,("Sblo_OpenVolumeHandle:
    | Error %08x in
    | ObReferenceObjectByHandle('%S')\n",Status,VolumeName));
42218|         ZwClose (VolumeHandle);
42219|         VolumeHandle = INVALID_HANDLE_VALUE;
42220|         VolumeFileObject = NULL;
42221|     }

```

```

42222| } else {
42223|     Debug(DEBUG_DEVSUP,("Sblo_OpenVolumeHandle:
| Error %08x in
| ZwCreateFile("%S")\n",Status,VolumeName));
42224|     VolumeHandle = INVALID_HANDLE_VALUE;
42225| }
42226|
42227| return Status;
42228| }
42229|
42230|
42231| NTSTATUS Sblo_CloseVolumeHandle (
42232|     HANDLE      &VolumeHandle,
42233|     PFILE_OBJECT &VolumeFileObject )
42234| {
42235|     NTSTATUS Status = STATUS_SUCCESS;
42236|     if ( VolumeFileObject==NULL ||
| !IsValidHandle(VolumeHandle) ) {
42237|         ASSERT ( VolumeFileObject != NULL );
42238|         ASSERT ( IsValidHandle(VolumeHandle) );
42239|         Status = STATUS_INVALID_PARAMETER;
42240|         Debug(DEBUG_DEVSUP,("Sblo_CloseVolumeHandle:
| Invalid parameter -- VolumeHandle=%08x,
| VolumeFileObject=%08x\n",VolumeHandle,VolumeFileObject))
| ;
42241|     } else {
42242|         ObDereferenceObject (VolumeFileObject);
42243|         ZwClose (VolumeHandle);
42244|
42245|         VolumeFileObject = NULL;
42246|         VolumeHandle = INVALID_HANDLE_VALUE;
42247|     }
42248|
42249| return Status;
42250| }
42251|
42252|
42253| NTSTATUS Sblo_SystemCall (
42254|     const WCHAR      *VolumeName,
42255|     FS_SYSTEM_CALL_FUNCTION  FsFunctionToCall,
42256|     const char      *ActionDebugText,
42257|     ULONG           DesiredAccess )
42258| {
42259|     HANDLE      FileHandle = INVALID_HANDLE_VALUE;
42260|     PFILE_OBJECT  FileObject = NULL;
42261|     Debug(DEBUG_DEVSUP,("Sblo_SystemCall: %sing volume
| '%S'\n",ActionDebugText,VolumeName));
42262|     NTSTATUS Status = Sblo_OpenVolumeHandle (
| VolumeName, FileHandle, FileObject, DesiredAccess );
42263|     if ( NT_SUCCESS(Status) ) {

```

```

42264|     Status = (*FsFunctionToCall) (FileObject);
42265|     if(NT_SUCCESS(Status)) {
42266|         Debug(DEBUG_DEVSUP,("Sblo_SystemCall:
| Success %sing volume
| '%S'\n",ActionDebugText,VolumeName));
42267|     } else {
42268|         Debug(DEBUG_DEVSUP,("Sblo_SystemCall: Error
| %08x %sing volume
| '%S'\n",Status,ActionDebugText,VolumeName));
42269|     }
42270|     Sblo_CloseVolumeHandle ( FileHandle, FileObject
| );
42271| }
42272|
42273| if ( !NT_SUCCESS(Status) ) {
42274|     Debug(DEBUG_DEVSUP,("Sblo_SystemCall:
| Returning %08x\n",Status));
42275| }
42276|
42277| return Status;
42278| }
42279|
42280|
42281| NTSTATUS Sblo_DismountVolume( const WCHAR *VolumeName )
42282| {
42283|     return Sblo_SystemCall ( VolumeName,
| FS_DismountVolume, "dismount", GENERIC_READ );
42284| }
42285|
42286|
42287|
42288|
42289| /*-----
| -----*/
42290| NTSTATUS Sblo_InvalidateVolumes ( PDEVICE_OBJECT
| FSObject, HANDLE FileHandle )
42291| {
42292|     PIRP                Irp=NULL;
42293|     KEVENT               Event={0};
42294|     IO_STATUS_BLOCK       IoStatusBlock={0};
42295|     NTSTATUS              Status=0;
42296|     PIO_STACK_LOCATION    IrpStack=NULL;
42297|
42298|     __try {
42299|
42300|         Debug(DEBUG_DEVSUP,("Sblo_InvalidateVolumes:
| invalidating volume %08x\n", FSObject));
42301|
42302|         //
42303|         // Set the event object to the unsigned

```

```

    | state.
42304|    // It will be used to signal request
    | completion.
42305|    //
42306|
42307|    KeInitializeEvent(&Event, NotificationEvent,
    | FALSE);
42308|
42309|
42310|    //
42311|    // Create IRP for read
42312|    //
42313|
42314|    Irp = IrpAllocateIrp(FSObject->StackSize);
42315|
42316|    if (!Irp) {
42317|
    | Debug(DEBUG_DEVSUP,("Sblo_InvalidateVolumes: Error!
    | Unable to allocate irp\n"));
42318|        try_return(Status =
    | STATUS_INSUFFICIENT_RESOURCES);
42319|    }
42320|
42321|    Irp->Flags = 0; //IRP_READ_OPERATION;
42322|
42323|    Irp->AssociatedIrp.SystemBuffer =
    | &FileHandle;
42324|    Irp->MdlAddress = NULL;
42325|    Irp->UserBuffer = NULL;
42326|    Irp->UserEvent = &Event;
42327|    Irp->UserIoSB = &IoStatusBlock;
42328|    /*lint -save -e740 */
42329|    Irp->Tail.Overlay.Thread =
    | PsGetCurrentThread();
42330|    /*lint -restore */
42331|    Irp->Tail.Overlay.OriginalFileObject = NULL;
42332|    Irp->RequestorMode =
    | (KPROCESSOR_MODE)KernelMode;
42333|    Irp->IoStatus.Status =
    | STATUS_PENDING;
42334|    Irp->IoStatus.Information = 0;
42335|
42336|    IrpStack = IoGetNextIrpStackLocation(Irp);
42337|    RtlZeroMemory((PVOID)IrpStack,
    | sizeof(IO_STACK_LOCATION));
42338|    // just sets event
42339|    IoSetCompletionRoutine(Irp,
    | Sblo_ReadDeviceMdlCompletionRoutine, NULL, TRUE, TRUE,
    | TRUE);
42340|    IrpStack->MajorFunction =

```

```

    | IRP_MJ_FILE_SYSTEM_CONTROL;
42341|     IrpStack->MinorFunction =
    | IRP_MN_USER_FS_REQUEST;
42342|
    | IrpStack->Parameters.DeviceIoControl.IoControlCode =
    | FSCTL_INVALIDATE_VOLUMES;
42343|
    | IrpStack->Parameters.DeviceIoControl.OutputBufferLength
    | = 0;
42344|
    | IrpStack->Parameters.DeviceIoControl.InputBufferLength
    | = sizeof(HANDLE);
42345|
    | IrpStack->Parameters.DeviceIoControl.Type3InputBuffer =
    | NULL;
42346|     IrpStack->DeviceObject = FSObject;
42347|     IrpStack->FileObject = NULL;
42348|
42349|     Status = IoCallDriver(FSObject, Irp);
42350|
42351|     if (Status == STATUS_PENDING) {
42352|
42353|
    | Debug(DEBUG_DEVSUP,("Sblo_InvalidateVolumes: Waiting
    | for dismount to finish\n"));
42354|     ASSERT(KeGetCurrentIrql() <
    | DISPATCH_LEVEL);
42355|     pmWaitForSingleObject(&Event,NULL);
42356|
42357|     Status = IoStatusBlock.Status;
42358|     }
42359|     IrpFreeIrp(Irp);
42360|
42361| try_exit: NOTHING;
42362|     }
    | __except(ExceptionFilter(GetExceptionInformation())) {
42363|     Status = GetExceptionCode();
42364|     Debug(DEBUG_DEVSUP,("Sblo_InvalidateVolumes:
    | Exception %08x\n",Status));
42365|     }
42366|
42367|     return Status;
42368| }
42369|
42370| /*
42371|  Several ways to get the FS object
42372|  1. first object created is usually the fs object
42373|  2. usually only the fs object has a name
42374|  */
42375| /*-----

```



```

| -----*/
42376| PDEVICE_OBJECT GetFSObjectFromVolumeObject (
| PDEVICE_OBJECT Volume )
42377| {
42378|     PDEVICE_OBJECT DevObj = Volume;
42379|
42380|     if(!DevObj) {
42381|         return NULL;
42382|     }
42383|
42384|     // get first object created (at end of linked list)
42385|     while(DevObj->NextDevice) {
42386|         DevObj = DevObj->NextDevice;
42387|     }
42388|
42389|     // okay, lets make sure it is named
42390|     if(DevObj->Flags & DO_DEVICE_HAS_NAME) {
42391|         return DevObj;
42392|     } else {
42393|         return NULL;
42394|     }
42395| }
42396|
42397| /*-----
| -----*/
42398| NTSTATUS InvalidateVolumes( PDEVICE_OBJECT Volume )
42399| {
42400|     HANDLE VolumeHandle=INVALID_HANDLE_VALUE;
42401|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
42402|     PDEVICE_OBJECT FsObject=NULL;
42403|     POBJECT_NAME_INFORMATION OBI=NULL;
42404|     PFILE_OBJECT FileObject=NULL;
42405|     PDEVICE_OBJECT DeviceObject=NULL;
42406|     ULONG Returned=0;
42407|     OBJECT_ATTRIBUTES ObjAttr={0};
42408|     IO_STATUS_BLOCK IoStatus={0};
42409|
42410|     // Volume = "virtual volume" by psm
42411|     // Volume->Vpb->DeviceObject == unnamed filesystem
| object
42412|     // Volume->Vpb->RealDevice == Volume as owned by
| lowest device (eg \harddisk0\partition1)
42413|
42414|     FsObject =
| GetFSObjectFromVolumeObject(Volume->Vpb->DeviceObject);
42415|
42416|     if(!FsObject) {
42417|         return STATUS_OBJECT_NAME_NOT_FOUND;
42418|     }
42419|

```

```

42420|  if((Volume->Vpb) && (Volume->Vpb->Flags &
    | VPB_MOUNTED)) {
42421|
42422|      // we need to get a FileObject to the "Volume"
42423|
42424|      // step 1. We need the device name
    | (\harddisk0\partition1)
42425|
42426|      Status = ObQueryNameString(
42427|          Volume->Vpb->RealDevice,
42428|          NULL, 0, &Returned );
42429|
42430|      if((Status==STATUS_INFO_LENGTH_MISMATCH) &&
    | (Returned>0)) {
42431|          OBI = (POBJECT_NAME_INFORMATION)
    | MemAllocatePoolWithTag(PagedPool,Returned,FILENAMETAG);
42432|          if(OBI) {
42433|
42434|              Status = ObQueryNameString(
42435|                  Volume->Vpb->RealDevice,
42436|                  OBI,
42437|                  Returned,
42438|                  &Returned );
42439|
42440|              if(NT_SUCCESS(Status)) {
42441|                  // step 2. get file object for
    | volume
42442|                  Status = IoGetDeviceObjectPointer(
42443|                      &OBI->Name,
42444|                      0,
42445|                      &FileObject,
42446|                      &DeviceObject
42447|                  );
42448|
42449|                  if(NT_SUCCESS(Status)) {
42450|                      // step 3. get handle for file
    | object
42451|                      /* fails with 0xc0000001 STATUS_UNSUCCESSFUL
42452|                      Status = ObOpenObjectByPointer(
42453|                          FileObject,
42454|                          0,
42455|                          NULL,
42456|                          0,
42457|                          *IoFileObjectType,
42458|
    | (KPROCESSOR_MODE)KernelMode,
42459|                          &VolumeHandle
42460|                      );
42461|
42462|                      */

```

```

42463|
42464|             InitializeObjectAttributes (
    | &ObjAttr,
42465|                 &OBI->Name,
    | OBJ_CASE_INSENSITIVE, NULL, NULL );
42466|
42467|             Status = ZwOpenFile(
42468|                 &VolumeHandle,
42469|                 0,
42470|                 &ObjAttr,
42471|                 &IoStatus,
42472|                 0,
42473|                 0
42474|             );
42475|
42476|             if(NT_SUCCESS(Status)) {
42477|                 // step 4. Tell file system
    | to invalidate
42478|
    | ASSERT(IsValidHandle(VolumeHandle));
42479|                 Status =
    | Sblo_InvalidateVolumes(FsObject,VolumeHandle);
42480|
42481|                 // close handle
42482|                 ZwClose(VolumeHandle);
42483|                 VolumeHandle =
    | INVALID_HANDLE_VALUE;
42484|             } else {
42485|
    | Debug(DEBUG_DEVSUP,("Devsup: InvalidateVolumes: Error
    | %08x getting handle\n",Status));
42486|             }
42487|             // close object
42488|
    | ObDereferenceObject(FileObject);
42489|             } else {
42490|                 Debug(DEBUG_DEVSUP,("Devsup:
    | InvalidateVolumes: Error %08x getting device object
    | pointer\n",Status));
42491|             }
42492|             } else {
42493|                 Debug(DEBUG_DEVSUP,("Query name
    | failed error %08x\n",Status));
42494|             }
42495|             FREE_POINTER(OBI);
42496|             } else {
42497|                 Debug(DEBUG_DEVSUP,("Out of memory for
    | %d byte\n",Returned));
42498|             }
42499|             } else {

```

```

42500|         Debug(DEBUG_DEVSUP,("Query name failed
      | getting size error %08x\n",Status));
42501|     }
42502|
42503| } else {
42504|     Debug(DEBUG_DEVSUP,("Devsup: InvalidateVolumes:
      | volume %08x not mounted\n",Volume));
42505| }
42506| return Status;
42507| }
42508|
42509|
42510| /*-----
      | -----*/
42511| NTSTATUS Sblo_GetGeometry( PDEVICE_OBJECT
      | DeviceObject,
42512|                             PDISK_GEOMETRY Geometry )
42513| {
42514|     PIRP                Irp=NULL;
42515|     KEVENT                Event={0};
42516|     IO_STATUS_BLOCK       IoStatusBlock={0};
42517|     NTSTATUS              Status=0;
42518|
42519|     __try {
42520|
42521|         //
42522|         // Set the event object to the unsigned
      | state.
42523|         // It will be used to signal request
      | completion.
42524|         //
42525|
42526|         KeInitializeEvent(&Event,
42527|                           NotificationEvent,
42528|                           FALSE);
42529|
42530|
42531|
42532|         //
42533|         // Create IRP for get drive layout device
      | control.
42534|         //
42535|
42536|         Irp =
      | IoBuildDeviceIoControlRequest(IOCTL_DISK_GET_DRIVE_GEOME
      | TRY,
42537|
      | DeviceObject,
42538|
      | NULL,
42539|
      | 0,

```

```

42540|                                     Geometry,
42541|
42542|                                     FALSE,
42543|                                     &Event,
42544|
42545|                                     | &IoStatusBlock);
42546|
42547|         if (!Irp) {
42548|             Debug(DEBUG_DEVSUP,("Sblo_GetDriveLayout:
42549|             | Error! Unable to allocate irp\n"));
42550|             return STATUS_INSUFFICIENT_RESOURCES;
42551|         }
42552|
42553|         Status = IoCallDriver(DeviceObject, Irp);
42554|
42555|         if (Status == STATUS_PENDING) {
42556|             ASSERT(KernelGetCurrentIrql() <
42557|             | DISPATCH_LEVEL);
42558|             KeWaitForSingleObject(&Event,NULL);
42559|
42560|             Status = IoStatusBlock.Status;
42561|         }
42562|     }
42563|     | __except(ExceptionFilter(GetExceptionInformation())) {
42564|         Status = GetExceptionCode();
42565|         Debug(DEBUG_DEVSUP,("Sblo_GetDriveLayout:
42566|         | Exception %08x\n",Status));
42567|     }
42568|     return Status;
42569| }
42570|
42571| /*-----*/
42572| | -----*/
42573| NTSTATUS Sblo_GetCapabilities( PDEVICE_OBJECT
42574| | DeviceObject,
42575| | PIO SCSI CAPABILITIES
42576| | Caps)
42577| {
42578|     PIRP             Irp=NULL;
42579|     KEVENT            Event={0};
42580|     IO_STATUS_BLOCK   IoStatusBlock={0};
42581|     NTSTATUS          Status=0;
42582|
42583|     __try {
42584|
42585|         //
42586|         // Set the event object to the unsignaled
42587|         | state.

```

```

42580|         // It will be used to signal request
         | completion.
42581|         //
42582|
42583|         KeInitializeEvent(&Event,
42584|             NotificationEvent,
42585|             FALSE);
42586|
42587|
42588|
42589|         //
42590|         // Create IRP for get drive layout device
         | control.
42591|         //
42592|
42593|         Irp =
         | IoBuildDeviceIoControlRequest(IOCTL_SCSI_GET_CAPABILITIE
         | S,
42594|         | DeviceObject,
42595|             NULL,
42596|             0,
42597|             Caps,
42598|
         | sizeof(IO_SCSI_CAPABILITIES),
42599|             FALSE,
42600|             &Event,
42601|         | &IoStatusBlock);
42602|
42603|         if (!Irp) {
42604|             Debug(DEBUG_DEVSUP,("SbloGetCapabilites:
         | Error! Unable to allocate irp\n"));
42605|             return STATUS_INSUFFICIENT_RESOURCES;
42606|         }
42607|
42608|         Status = IoCallDriver(DeviceObject, Irp);
42609|
42610|         if (Status == STATUS_PENDING) {
42611|
42612|             ASSERT(KeGetCurrentIrql() <
         | DISPATCH_LEVEL);
42613|             pmWaitForSingleObject(&Event, NULL);
42614|
42615|             Status = IoStatusBlock.Status;
42616|         }
42617|     }
         | __except(ExceptionFilter(GetExceptionInformation())) {
42618|         Status = GetExceptionCode();
42619|         Debug(DEBUG_DEVSUP,("Sblo_GetCapabilities:

```

```

    | Exception %08x\n",Status));
42620|    }
42621|    return Status;
42622| }
42623|
42624|
42625| /*-----
    | -----*/
42626| NTSTATUS Sblo_GetDriveLayout( PDEVICE_OBJECT
    | DeviceObject,
42627|
    | PDRIVE_LAYOUT_INFORMATION DriveLayoutInfo,
42628|          ULONG
    | DriveLayoutInfoSize )
42629| {
42630|     PIRP          Irp=NULL;
42631|     KEVENT         Event={0};
42632|     IO_STATUS_BLOCK IoStatusBlock={0};
42633|     NTSTATUS       Status=0;
42634|
42635|     __try {
42636|
42637|         //
42638|         // Set the event object to the unsigned
    | state.
42639|         // It will be used to signal request
    | completion.
42640|         //
42641|
42642|         KeInitializeEvent(&Event,
42643|             NotificationEvent,
42644|             FALSE);
42645|
42646|
42647|         //
42648|         // Create IRP for get drive layout device
    | control.
42649|         //
42650|
42651|         Irp =
    | IoBuildDeviceIoControlRequest(IOCTL_DISK_GET_DRIVE_LAYOUT
    | T,
42652|
    | DeviceObject,
42653|             NULL,
42654|             0,
42655|
    | DriveLayoutInfo,
42656|
    | DriveLayoutInfoSize,

```

```

42657|                 FALSE,
42658|                 &Event,
42659|                 | &IoStatusBlock);
42660|
42661|         if (!Irp) {
42662|             Debug(DEBUG_DEVSUP,("Sblo_GetDriveLayout:
42663|             | Error! Unable to allocate irp\n"));
42664|             return STATUS_INSUFFICIENT_RESOURCES;
42665|         }
42666|
42667|         Status = IoCallDriver(DeviceObject, Irp);
42668|
42669|         if (Status == STATUS_PENDING) {
42670|
42671|             ASSERT(KeGetCurrentIrql() <
42672|             | DISPATCH_LEVEL);
42673|             Debug(DEBUG_DEVSUP,("Sblo_GetDriveLayout:
42674|             | Waiting for request to finish\n"));
42675|             pmWaitForSingleObject(&Event,NULL);
42676|
42677|             Status = IoStatusBlock.Status;
42678|         } else
42679|         if(Status!=STATUS_SUCCESS) {
42680|             Debug(DEBUG_DEVSUP,("Sblo_GetDriveLayout:
42681|             | Error %08x reading drive layout\n",Status));
42682|         }
42683|     }
42684| }
42685|
42686|     | __except(ExceptionFilter(GetExceptionInformation())) {
42687|         Status = GetExceptionCode();
42688|         Debug(DEBUG_DEVSUP,("Sblo_GetDriveLayout:
42689|         | Exception %08x\n",Status));
42690|     }
42691| }
42692|
42693| return Status;
42694|
42695| }
42696|
42697| /*-----*/
42698| | -----*/
42699| NTSTATUS Sblo_DeviceIoControl( PDEVICE_OBJECT
42700| | DeviceObject,
42701| |         ULONG IoctlCode,
42702| |         char *InBuffer,
42703| |         ULONG InBufferSize,
42704| |         char *OutBuffer,
42705| |         ULONG OutBufferSize,
42706| |         ULONG *BytesReturned
42707| |         )

```



```

42698| {
42699|     PIRP Irp=NULL;
42700|     IO_STATUS_BLOCK iosb={0};
42701|     KEVENT Event={0};
42702|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
42703|
42704|     *BytesReturned = 0;
42705|
42706|     // Set the event object to the unsigaled state.
42707|     // It will be used to signal request completion.
42708|
42709|     KeInitializeEvent(&Event, NotificationEvent,
42710|         | FALSE);
42711|     Irp = IoBuildDeviceIoControlRequest( IoctlCode,
42712|         DeviceObject,
42713|         InBuffer,
42714|         InBufferSize,
42715|         OutBuffer,
42716|         OutBufferSize,
42717|         FALSE,
42718|         &Event,
42719|         &iosb);
42720|
42721|
42722|     if(Irp) {
42723|
42724|         __try {
42725|             Status = IoCallDriver(DeviceObject, Irp);
42726|
42727|             if (Status == STATUS_PENDING) {
42728|
42729|                 ASSERT(KeGetCurrentIrql() <
42730|                     | DISPATCH_LEVEL);
42731|                 pmWaitForSingleObject(&Event,NULL);
42732|                 Status = iosb.Status;
42733|             }
42734|         }
42735|         | __except(ExceptionFilter(GetExceptionInformation())) {
42736|             Status = GetExceptionCode();
42737|
42738|             | Debug(DEBUG_DEVSUP,("Sblo_DeviceIoControl:Error!
42739|             | Exception %08x sending scsi command\n",Status));
42740|         }
42741|     } else {
42742|         Debug(DEBUG_INFO,("Sblo_DeviceIoControl: Error
42743|         | no Irp available\n"));
42744|         Status = iosb.Status;
42745|     }

```

```

42742|
42743|   if (!INT_SUCCESS(Status)) {
42744|       *BytesReturned = losb.Information;
42745|   }
42746|
42747|
42748|   return Status;
42749| }
42750|
42751|
42752| #ifdef DEBUG
42753| /*-----
    | -----*/
42754| void Debug_DumpSector( char *Buffer, ULONG Size )
42755| {
42756|     ULONG i,j;
42757|     UCHAR *UBuffer = (UCHAR*)Buffer;
42758|     char s[80];
42759|     char *t;
42760|
42761| // 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
    | .....
42762|
42763| // no need to even try if debugging is not on..
42764| if(!DebugLevel) {
42765|     return;
42766| }
42767|
42768| for(i=0;i<Size;i+=16) {
42769|
42770|     t=s;
42771|
42772|     t+=sprintf(t,"%03x : ",i);
42773|
42774|     for(j=i;j-i<16;j++) {
42775|         t+=sprintf(t,"%02x ",UBuffer[j]);
42776|     }
42777|
42778|     for(j=i;j-i<16;j++) {
42779|         if ((UBuffer[j]>31) && (UBuffer[j]<128)) {
42780|             *t++ = UBuffer[j];
42781|         } else {
42782|             *t++ = '.';
42783|         }
42784|     }
42785|     *t=0;
42786|     Debug(DEBUG_INFO,("%s\n",s));
42787| }
42788| }
42789| #endif

```

```

42790|
42791| /*-----
| -----*/
42792| NTSTATUS CheckMediaLoaded ( PDEVICE_OBJECT
| DeviceObject, PIRP Irp )
42793| {
42794|     PVDISK_EXTENSION DevExt=NULL;
42795|     PIO_STACK_LOCATION irpSp=NULL;
42796|     NTSTATUS Err=STATUS_SUCCESS;
42797|
42798|     __try {
42799|         DevExt = GetVDiskExtension(DeviceObject);
42800|         irpSp = IoGetCurrentIrpStackLocation( Irp );
42801|
42802|         if ((!DevExt->PartitionActive) ||
| (DevExt->DriveNotReady)) {
42803|             // If the volume is mounted, we must tell
| the filesystem to
42804|             // verify that the media in the drive is
| the same volume.
42805|             //Debug(DEBUG_DEVSUP,("PSMan: Scsi: CML:
| Drive not ready\n"));
42806|
42807|             if ( DevExt->DeviceObject->Vpb->Flags &
| VPB_MOUNTED ) {
42808|                 //Debug(DEBUG_DEVSUP,("PSMan: Scsi:
| CML: Informing File system\n"));
42809|                 DevExt->DeviceObject->Flags |=
| DO_VERIFY_VOLUME;
42810|             }
42811|
| InterlockedIncrement((PLONG)&DevExt->DiskChangeCount);
42812|         }
42813|
42814|         if ( (DevExt->DeviceObject->Flags &
| DO_VERIFY_VOLUME) &&
42815|             !(irpSp->Flags & SL_OVERRIDE_VERIFY_VOLUME)
42816|         ) {
42817|
42818|             //Debug(DEBUG_DEVSUP,("PSMan: Scsi: CML:
| Verify bit is set\n"));
42819|
42820|             // if DO_VERIFY_VOLUME bit is set
42821|             // in device object flags, fail request.
42822|             IoSetHardErrorOrVerifyDevice(Irp,
| DeviceObject);
42823|
42824|             Irp->IoStatus.Status =
| STATUS_VERIFY_REQUIRED;
42825|             Irp->IoStatus.Information = 0;

```

```

42826|
42827|     Err = Irp->IoStatus.Status;
42828| } else {
42829|     // if we are still here, then just say
    | error..
42830|     if ((!DevExt->PartitionActive) ||
    | (DevExt->DriveNotReady)) {
42831|         Irp->IoStatus.Status =
    | STATUS_IO_DEVICE_ERROR; //STATUS_NO_MEDIA_IN_DEVICE;
42832|         Irp->IoStatus.Information = 0;
42833|         Err = Irp->IoStatus.Status;
42834|     }
42835| }
42836| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
42837|     Irp->IoStatus.Status = GetExceptionCode();
42838|
42839|     Debug(DEBUG_DEVSUP,("PSMan: Scsi: CML:
    | Exception %08x\n",Irp->IoStatus.Status));
42840|     Err = Irp->IoStatus.Status;
42841| }
42842|
42843| return Err;
42844| }
42845|
42846| // pass in -1 for thread to get any user for a process.
42847| /*-----*/
    | -----*/
42848| pOT_USER FindPSMUser ( PEPROCESS Process, PETHREAD
    | Thread )
42849| {
42850|     pOT_USER Next=NULL;
42851|     PAGED_CODE();
42852|
42853|     pmAcquireMutex ( &PSMUserMutex, NULL );
42854|     Next=GlobalData->PSMUsers;
42855|     while(Next) {
42856|         if ( Next->ProcessID == Process ) {
42857|             if ( Thread==(PETHREAD)-1 ||
    | Thread==Next->ThreadID ) {
42858|                 break;
42859|             }
42860|         }
42861|
42862|         Next = Next->Next;
42863|     }
42864|     pmReleaseMutex ( &PSMUserMutex );
42865|
42866|     return Next;
42867| }

```

```

42868|
42869| // pass in -1 for thread to get any user for a process.
42870| /*-----
    | -----*/
42871| pOT_USER FindPSMUserByFileObject ( PFILE_OBJECT
    | FileObject )
42872| {
42873|     pOT_USER Next=NULL;
42874|     PAGED_CODE();
42875|
42876|     pmAcquireMutex ( &PSMUserMutex, NULL );
42877|     Next=GlobalData->PSMUsers;
42878|     while(Next) {
42879|         if(FileObject==Next->FileObject) {
42880|             break;
42881|         }
42882|
42883|         Next = Next->Next;
42884|     }
42885|     pmReleaseMutex ( &PSMUserMutex );
42886|
42887|     return Next;
42888| }
42889|
42890| /*-----
    | -----*/
42891| pOT_USER FindPSMUserBySnapShot ( pkSnapShotMaster
    | SnapShot )
42892| {
42893|     pOT_USER Next=NULL;
42894|     pkSnapShotEntry p;
42895|
42896|     PAGED_CODE();
42897|
42898|
42899|     GetSnapShotForRead();
42900|     __try {
42901|
    | p=GetTopSnapShotForMaster(&SnapShot->SnapShots);
42902|         while(p) {
42903|             pmAcquireMutex ( &PSMUserMutex, NULL );
42904|             __try {
42905|                 Next=GlobalData->PSMUsers;
42906|                 while(Next) {
42907|
    | if(IsInUserList(Next,p)==STATUS_SUCCESS) {
42908|                     break;
42909|                 }
42910|
    |
42911|                 Next = Next->Next;

```

```

42912|         }
42913|     } __finally {
42914|         pmReleaseMutex ( &PSMUserMutex );
42915|     }
42916|     if(Next) {
42917|         // found one
42918|         DoneWithSnapShot(p);
42919|         break;
42920|     } else {
42921|
42922|         | p=GetNextSnapShotForMaster(&SnapShot->SnapShots,p);
42923|     }
42924| } __finally {
42925|     ReleaseSnapShotForRead();
42926| }
42927|
42928| return Next;
42929| }
42930|
42931|
42932| /*-----
42933| | -----*/
42934| void AddPSMUser( pOT_USER User )
42935| {
42936|     PAGED_CODE();
42937|     pmAcquireMutex ( &PSMUserMutex, NULL );
42938|     User->Next = GlobalData->PSMUsers;
42939|     GlobalData->PSMUsers = User;
42940|     pmReleaseMutex ( &PSMUserMutex );
42941| }
42942|
42943| /*-----
42944| | -----*/
42945| void DeletePSMUser( pOT_USER User )
42946| {
42947|     pOT_USER Prev=NULL;
42948|     pOT_USER Next=NULL;
42949|     PAGED_CODE();
42950|     pmAcquireMutex ( &PSMUserMutex, NULL );
42951|     Next=GlobalData->PSMUsers;
42952|     while(Next) {
42953|         if(Next==User) {
42954|             if(Prev==NULL) {
42955|                 GlobalData->PSMUsers = Next->Next;
42956|             } else {
42957|                 Prev->Next = Next->Next;
42958|             }

```

```

42959|
42960|         break;
42961|     }
42962|     Prev = Next;
42963|     Next = Next->Next;
42964| }
42965| pmReleaseMutex ( &PSMUserMutex );
42966| }
42967|
42968| /*-----
| -----*/
42969| NTSTATUS FlushVolume( const WCHAR *Name )
42970| {
42971|     UNICODE_STRING  UniName={0};
42972|     OBJECT_ATTRIBUTES ObjectAttributes={0};
42973|     NTSTATUS        Status=STATUS_UNSUCCESSFUL;
42974|     IO_STATUS_BLOCK IoStatus={0};
42975|     HANDLE          FileHandle=NULL;
42976|
42977|     PAGED_CODE();
42978|
42979|     RtlInitUnicodeString (&UniName, Name);
42980|
42981|
42982|     InitializeObjectAttributes ( &ObjectAttributes,
42983|                                  &UniName,
42984|                                  OBJ_CASE_INSENSITIVE,
42985|                                  NULL,
42986|                                  NULL );
42987|
42988| DoAgain:
42989|     Status = ZwOpenFile( &FileHandle,
42990|                          FILE_READ_DATA |
42991|                          | FILE_WRITE_DATA | SYNCHRONIZE, // desired access
42992|                          &ObjectAttributes, //
42993|                          | object attributes
42994|                          &IoStatus,
42995|                          FILE_SHARE_WRITE |
42996|                          | FILE_SHARE_READ, // share access
42997|                          | FILE_SYNCHRONOUS_IO_NONALERT ); // open options
42998|
42999|     if(NT_SUCCESS(Status)) {
43000| Flush:
43001|         Status =
43002|         | MyFlushBuffersFile(FileHandle,&IoStatus);
43003|
43004|         if(!NT_SUCCESS(Status)) {
43005|             Debug(DEBUG_DEVSUP,("Devsup: flushVolumes:
43006|             | Error %08x flushing\n",Status));

```

```

43002|     }
43003|
43004|     if(Status==STATUS_PENDING) {
43005|         pmWaitForSingleObject(FileHandle,NULL);
43006|
43007|         Status = IoStatus.Status;
43008|     }
43009|     if(!NT_SUCCESS(Status)) {
43010|         Debug(DEBUG_DEVSUP,("PSMan: FlushVolume:
| Error %08x flshing volume '%S'\n",Status,Name));
43011|     }
43012|     ZwClose(FileHandle);
43013|     FileHandle=NULL;
43014| } else {
43015|     Debug(DEBUG_DEVSUP,("PSMan: FlushVolume: Error
| %08x opening volume '%S'\n",Status,Name));
43016|     // volume got mounted, try again
43017|     if(Status==STATUS_REPARSE) {
43018|         goto DoAgain;
43019|     }
43020|
43021|     // being handled async..
43022|     if(Status==STATUS_PENDING) {
43023|         Debug(DEBUG_DEVSUP,("PSMan: FlushVolume:
| Waiting for file %d to open\n",FileHandle));
43024|         pmWaitForSingleObject(FileHandle,NULL);
43025|
43026|         Status = IoStatus.Status;
43027|         if(NT_SUCCESS(Status)) {
43028|             goto Flush;
43029|         } else {
43030|             Debug(DEBUG_DEVSUP,("PSMan:
| FlushVolume: Error %08x opening volume
| '%S'\n",Status,Name));
43031|             ZwClose(FileHandle);
43032|             FileHandle=NULL;
43033|         }
43034|     }
43035| }
43036|
43037| return Status;
43038| }
43039|
43040| #ifdef DEBUG
43041|
43042| STATIC ULONG CurrentDebugLine=0;
43043| STATIC char Lines[MAX_BACK_LOG_FOR_DEBUG][255]={0};
43044|
43045| /*-----
| -----*/

```



```

43046| void DebugPrintSave( char *fmt, ... )
43047| {
43048|     va_list argptr;
43049|     LARGE_INTEGER Time;
43050|     ULONG Len;
43051|     KIRQL oldIrql;
43052|     ULONG OurLine;
43053|     LARGE_INTEGER Ticks;
43054|
43055|     // do not use pm* functions in here
43056|     // or call any other function that does
43057|     // or a deadlock will occur
43058|     // we are imposing this limit so the pm*
43059|     // functions can write to the debug log file
43060|
43061|     ASSERT(CurrentDebugLine<MAX_BACK_LOG_FOR_DEBUG);
43062|     KeAcquireSpinLock( &PSM_DebugSpinLock, &oldIrql );
43063|
43064|     OurLine = CurrentDebugLine++;
43065|     if(CurrentDebugLine>=MAX_BACK_LOG_FOR_DEBUG) {
43066|         CurrentDebugLine = 0;
43067|     }
43068|     KeReleaseSpinLock( &PSM_DebugSpinLock, oldIrql );
43069|
43070|     va_start(argptr, fmt);
43071|     vsprintf(Lines[OurLine], fmt, argptr);
43072|     va_end(argptr);
43073|
43074|     KeQuerySystemTime(&Time);
43075|     Ticks = KeQueryPerformanceCounter( NULL );
43076|     DbgPrint("%08x%08x %08x%08x (%08x):
| %s",Time.HighPart,Time.LowPart,Ticks.HighPart,Ticks.LowP
| art,PsGetCurrentThread(),Lines[OurLine]);
43077| #ifdef DEBUG
43078|     Len = strlen(Lines[OurLine]);
43079|
43080|     if(Len>200) {
43081|         DbgPrint("Above Line length = %d >
| 200!!!",Len);
43082|         DbgBreakPoint();
43083|     }
43084| #endif
43085|     Lines[OurLine][200] = 0;
43086|
43087|     if(gDebugToLog) {
43088|         // we dont use MemAllocate* here because we
| dont want this information logged
43089|         tDebugLogEntry *DebugEntry = (tDebugLogEntry *)
| ExAllocatePoolWithTag(NonPagedPool,sizeof(tDebugLogEntry
| ),DEBUG_ENTRY_TAG);

```

```

43090|     if(DebugEntry) {
43091|         sprintf(DebugEntry->LogEntry,"%08x%08x
| %08x%08x (%08x):
| %s",Time.HighPart,Time.LowPart,Ticks.HighPart,Ticks.LowP
| art,PsGetCurrentThread(),Lines[OurLine]);
43092|
43093|         ExInterlockedInsertTailList (
| &PSM_DebugQueue,
43094|         | &(DebugEntry->ListEntry),
43095|         | &PSM_DebugSpinLock);
43096|
43097|         KeReleaseSemaphore(
| &PSM_DebugSemaphore,1,1,FALSE);
43098|     } else {
43099|         DbgPrint("Out of memory for line
| '%s'\n",Lines[OurLine]);
43100|     }
43101| }
43102| }
43103|
43104| /*-----
| -----*/
43105| void BugCheckCallBack( PVOID Buffer, ULONG /*Length*/ )
43106| {
43107|     ULONG i,j;
43108|
43109|     // display to screen
43110|     i=MAX_BACK_LOG_FOR_DEBUG;
43111|
43112|     if(CurrentDebugLine>0)
43113|         j=CurrentDebugLine-1;
43114|     else
43115|         j=MAX_BACK_LOG_FOR_DEBUG-1;
43116|
43117|     while(i) {
43118|         HalDisplayString(Lines[j]);
43119|         if(j>0)
43120|             j--;
43121|         else
43122|             j=MAX_BACK_LOG_FOR_DEBUG-1;
43123|
43124|         i--;
43125|     }
43126|
43127|     // store in bugcheck data
43128|     memmove(Buffer,Lines,MAX_BACK_LOG_FOR_DEBUG*255);
43129|
43130|     // loop forever so user can read screen to us

```

```

43131|   while(1);
43132| }
43133|
43134| /*-----
    | -----*/
43135| void DebugLogThread( PVOID Context )
43136| {
43137|   UNICODE_STRING Uni;
43138|   UNICODE_STRING LogFile;
43139|   WCHAR Buff[256];
43140|   OBJECT_ATTRIBUTES ObjectAttributes;
43141|   IO_STATUS_BLOCK IoStatus;
43142|   HANDLE FileHandle;
43143|   tDebugLogEntry *DebugEntry;
43144|   NTSTATUS Status;
43145|   PLIST_ENTRY ListEntry;
43146|   LARGE_INTEGER TimeToWait;
43147|
43148|   Reg_GetStringKey (
    | &gRegistryPath,L"DebugLogFile",L"C:\\psm.log",&Uni);
43149|
43150|   LogFile.Buffer = Buff;
43151|   LogFile.Length = 0;
43152|   LogFile.MaximumLength = 256*2;
43153|
43154|   | RtlAppendUnicodeToString(&LogFile,L"\\DosDevices\\");
43155|   RtlAppendUnicodeStringToString (&LogFile,&Uni);
43156|   Reg_FreeString(&Uni);
43157|
43158|   InitializeObjectAttributes ( &ObjectAttributes,
43159|                               &LogFile,
43160|                               OBJ_CASE_INSENSITIVE,
43161|                               NULL,
43162|                               NULL );
43163|
43164| TryOpenAgain:
43165|   Status = ZwCreateFile( &FileHandle,
43166|                         FILE_GENERIC_READ |
    | FILE_GENERIC_WRITE, // desired access
43167|                         &ObjectAttributes, // object
    | attributes
43168|                         &IoStatus,
43169|                         NULL,          //
    | alloc size
43170|                         FILE_ATTRIBUTE_NORMAL,
    | // file attributes
43171|                         FILE_SHARE_READ,
    | // share access
43172|                         FILE_OPEN, //

```

```

| FILE_OVERWRITE_IF,          // create
| disposition
43173|
| FILE_SYNCHRONOUS_IO_NONALERT, // | FILE_WRITE_THROUGH,
| // create options
43174|          NULL, // eabuffer
43175|          0 ); // ealength
43176|
43177| if(Status!=STATUS_OBJECT_PATH_NOT_FOUND) {
43178|     if(NT_SUCCESS(Status)) {
43179|         PFILE_RENAME_INFORMATION Rename;
43180|         WCHAR *p;
43181|         UNICODE_STRING Uni;
43182|
43183|         DbgPrint("Renaming file\n");
43184|         Rename = (PFILE_RENAME_INFORMATION)
| MemAllocatePoolWithTag(PagedPool,sizeof(FILE_RENAME_INFO
| RMATION)+255,TEMPTAG);
43185|         Rename->ReplaceIfExists = TRUE;
43186|         Rename->RootDirectory = NULL;
43187|         p=wcsrchr(LogFile.Buffer,L'\\');
43188|         if(p) {
43189|             Uni.Buffer=Rename->FileName;
43190|             wcscpy(Rename->FileName,p+1);
43191|             wscat(Rename->FileName,L".bak");
43192|             Rename->FileNameLength = Uni.Length =
| (USHORT)NumBytes(Rename->FileName);
43193|             Uni.MaximumLength = 512;
43194|
43195|             Status = ZwSetInformationFile(
43196|                 FileHandle,          // IN
| HANDLE FileHandle,
43197|                 &IoStatus,          // OUT
| PIO_STATUS_BLOCK IoStatusBlock,
43198|                 Rename,              // IN PVOID
| FileInformation,
43199|
| sizeof(FILE_RENAME_INFORMATION)+255,    // IN
| ULONG Length,
43200|                 FileRenameInformation // IN
| FILE_INFORMATION_CLASS FileInformationClass
43201|             );
43202|             if(NT_SUCCESS(Status)) {
43203|                 DbgPrint("Success on renaming file
| to '%wZ'\n",&Uni);
43204|             } else {
43205|                 DbgPrint("Error %08x on renaming
| file to '%wZ'\n",Status,&Uni);
43206|             }
43207|         }

```

```

43208|         MemFreePool(Rename);
43209|         ZwClose(FileHandle);
43210|     }
43211| } else {
43212|     goto WaitFort;
43213| }
43214|
43215| Status = ZwCreateFile( &FileHandle,
43216|         FILE_GENERIC_READ |
43217|         | FILE_GENERIC_WRITE, // desired access
43218|         &ObjectAttributes, // object
43219|         | attributes
43220|         &IoStatus,
43221|         NULL, //
43222|         | alloc size
43223|         FILE_ATTRIBUTE_NORMAL,
43224|         | // file attributes
43225|         FILE_SHARE_READ,
43226|         | // share access
43227|         FILE_SUPERSEDE, //
43228|         | FILE_OVERWRITE_IF, // create
43229|         | disposition
43230|         | FILE_SYNCHRONOUS_IO_NONALERT, //| FILE_WRITE_THROUGH,
43231|         | // create options
43232|         NULL, // eabuffer
43233|         0 ); // ealength
43234|
43235| if(NT_SUCCESS(Status)) {
43236|
43237|     while(1) {
43238|
43239|         | pmAcquireSemaphore(&PSM_DebugSemaphore,NULL);
43240|
43241|         ListEntry = ExInterlockedRemoveHeadList (
43242|             &PSM_DebugQueue, // List Head
43243|             &PSM_DebugSpinLock // Lock
43244|         );
43245|
43246|         if((ListEntry) &&
43247|             | (ListEntry!=&PSM_DebugQueue)) {
43248|             /*lint -save -e413 */
43249|             DebugEntry = CONTAINING_RECORD(
43250|                 | ListEntry, tDebugLogEntry, ListEntry );
43251|             /*lint -restore */
43252|
43253|             Status = ZwWriteFile(
43254|                 FileHandle, // IN HANDLE
43255|                 | FileHandle,
43256|                 NULL, // IN HANDLE Event,

```

```

    | optional
43245|          NULL,          // IN
    | PIO_APC_ROUTINE ApcRoutine,          optional
43246|          NULL,          // IN PVOID
    | ApcContext,          optional
43247|          &IoStatus,    // OUT
    | PIO_STATUS_BLOCK IoStatusBlock,
43248|          DebugEntry->LogEntry,    // IN
    | PVOID Buffer,
43249|          strlen(DebugEntry->LogEntry),
    | // IN ULONG Length,
43250|          NULL,          // IN PLARGE_INTEGER
    | ByteOffset,          optional
43251|          NULL          // IN PULONG Key
    | optional
43252|          );
43253|
43254|          ExFreePool(DebugEntry);
43255|      }
43256|  }
43257|      // never get here because of while(1)
43258|      //ZwClose(FileHandle);
43259|
43260|  } else {
43261|  WaitForIt:
43262|      DbgPrint("Error %08x opening debug log file,
    | Waiting...\n",Status);
43263|      TimeToWait.QuadPart = RELATIVE(SECONDS(10));
43264|      KeDelayExecutionThread(
43265|          (KPROCESSOR_MODE)KernelMode, // IN
    | KPROCESSOR_MODE WaitMode,
43266|          FALSE,    // IN BOOLEAN Alertable,
43267|          &TimeToWait // IN PLARGE_INTEGER Interval
43268|      );
43269|      goto TryOpenAgain;
43270|  }
43271|
43272|  PsTerminateSystemThread( 0 );
43273| }
43274|
43275|
43276| #endif
43277|
43278| /*-----*/
    | -----*/
43279| pkSnapShotEntry GetTopSnapShot( PLIST_ENTRY ListHead )
43280| {
43281|     PLIST_ENTRY ListEntry;
43282|     pkSnapShotEntry p;
43283|

```

```

43284| #ifdef DEBUG
43285|     if((!IsSnapShotAcquiredForWrite()) &&
        | (!IsSnapShotAcquiredForRead())) {
43286|         Debug(DEBUG_DCPSM,("GetTopSnapShot: Snapshot
        | resource not acquired!\n"));
43287|         DbgBreakPoint();
43288|     }
43289| #endif
43290|
43291|     ListEntry = ListHead->Flink;
43292|
43293|     if((!ListEntry) || (ListEntry==ListHead)) {
43294| //         Debug(DEBUG_THREAD,("GetTopSnapShot:
        | ListEntry is empty!\n"));
43295|         return NULL;
43296|     }
43297|
43298|     /*lint -save -e413 */
43299|     p = (CONTAINING_RECORD( ListEntry, tkSnapShotEntry,
        | DevExt));
43300|     /*lint -restore */
43301|     UseSnapShot(p);
43302|     return p;
43303| }
43304|
43305| /*-----
        | -----*/
43306| pkSnapShotEntry GetNextSnapShot( PLIST_ENTRY ListHead,
        | pkSnapShotEntry SnapShot )
43307| {
43308|     PLIST_ENTRY ListEntry;
43309|     pkSnapShotEntry p;
43310|
43311| #ifdef DEBUG
43312|     if((!IsSnapShotAcquiredForWrite()) &&
        | (!IsSnapShotAcquiredForRead())) {
43313|         Debug(DEBUG_DCPSM,("GetNextSnapShot: Snapshot
        | resource not acquired!\n"));
43314|         DbgBreakPoint();
43315|     }
43316| #endif
43317|
43318|     ListEntry = SnapShot->DevExt.Flink;
43319|
43320|     if((!ListEntry) || (ListEntry==ListHead)) {
43321|         // end of list
43322|         DoneWithSnapShot(SnapShot);
43323|         return NULL;
43324|     }
43325|

```

```

43326|  /*lint -save -e413 */
43327|  p = (CONTAINING_RECORD( ListEntry, tkSnapshotEntry,
    | DevExt));
43328|  /*lint -restore */
43329|  UseSnapshot(p);
43330|  DoneWithSnapshot(Snapshot);
43331|  return p;
43332| }
43333|
43334| /*-----
    | -----*/
43335| pkSnapshotEntry GetTopSnapshotForMaster( PLIST_ENTRY
    | ListHead )
43336| {
43337|  PLIST_ENTRY ListEntry;
43338|  pkSnapshotEntry p;
43339|
43340| #ifdef DEBUG
43341|  if((!IsSnapshotAcquiredForWrite()) &&
    | (!IsSnapshotAcquiredForRead())) {
43342|      Debug(DEBUG_DCPSM,("GetTopSnapshotForMaster:
    | Snapshot resource not acquired!\n"));
43343|      DbgBreakPoint();
43344|  }
43345| #endif
43346|
43347|  ListEntry = ListHead->Flink;
43348|
43349|  if((!ListEntry) || (ListEntry==ListHead)) {
43350|  //      Debug(DEBUG_THREAD,("GetTopSnapshotForMaster:
    | ListEntry is empty\n"));
43351|      return NULL;
43352|  }
43353|
43354|  /*lint -save -e413 */
43355|  p = (CONTAINING_RECORD( ListEntry, tkSnapshotEntry,
    | Master));
43356|  /*lint -restore */
43357|  UseSnapshot(p);
43358|  return p;
43359| }
43360|
43361| /*-----
    | -----*/
43362| pkSnapshotEntry GetNextSnapshotForMaster( PLIST_ENTRY
    | ListHead, pkSnapshotEntry Snapshot )
43363| {
43364|  PLIST_ENTRY ListEntry;
43365|  pkSnapshotEntry p;
43366|

```



```

43367| #ifdef DEBUG
43368|     if((!IsSnapShotAcquiredForWrite()) &&
        | (!IsSnapShotAcquiredForRead())) {
43369|         Debug(DEBUG_DCPSM,("GetNextSnapShotForMaster:
        | Snapshot resource not acquired!\n"));
43370|         DbgBreakPoint();
43371|     }
43372| #endif
43373|
43374|     ListEntry = SnapShot->Master.Flink;
43375|
43376|     if((!ListEntry) || (ListEntry==ListHead)) {
43377|         // end of list
43378|         DoneWithSnapShot(SnapShot);
43379|         return NULL;
43380|     }
43381|
43382|     /*lint -save -e413 */
43383|     p = (CONTAINING_RECORD( ListEntry, tkSnapShotEntry,
        | Master));
43384|     /*lint -restore */
43385|     UseSnapShot(p);
43386|     DoneWithSnapShot(SnapShot);
43387|     return p;
43388| }
43389|
43390|
43391| /*-----
        | -----*/
43392| BOOLEAN InList( PLIST_ENTRY ListHead, tkSnapShotEntry
        | *SnapShot)
43393| {
43394|     tkSnapShotEntry *Entry;
43395|     PLIST_ENTRY Next = ListHead->Flink;
43396|
43397| #ifdef DEBUG
43398|     if((!IsSnapShotAcquiredForWrite()) &&
        | (!IsSnapShotAcquiredForRead())) {
43399|         Debug(DEBUG_DCPSM,("InList: Snapshot resource
        | not acquired!\n"));
43400|         DbgBreakPoint();
43401|     }
43402| #endif
43403|
43404|     if(!IsListEmpty(ListHead)) {
43405|         while(Next!=ListHead) {
43406|             Entry =
        | CONTAINING_RECORD(Next,tkSnapShotEntry,DevExt);
43407|             if(Entry == SnapShot) {
43408|                 return TRUE;

```

```

43409|     }
43410|     Next = Next->Flink;
43411|     }
43412| } else {
43413|     // list is empty so it cant be on it..
43414| }
43415| return FALSE;
43416| }
43417|
43418| /*-----
| -----*/
43419| pkSnapshotEntry
| FindSnapshotEntryForDevice(pkSnapshotMaster
| MasterSnapshot, PDEVICE_OBJECT DeviceObject)
43420| {
43421|     pkSnapshotEntry p=NULL;
43422|
43423| #ifdef DEBUG
43424|     if((!IsSnapshotAcquiredForWrite()) &&
| (!IsSnapshotAcquiredForRead())) {
43425|         Debug(DEBUG_DCPSM,("FindSnapshotEntryForDevice:
| Snapshot resource not acquired!\n"));
43426|         DbgBreakPoint();
43427|     }
43428| #endif
43429|
43430|     | p=GetTopSnapshotForMaster(&MasterSnapshot->SnapShots);
43431|     while(p) {
43432|         if(p->DeviceObject==DeviceObject) {
43433|             DoneWithSnapshot(p);
43434|             break;
43435|         }
43436|
| p=GetNextSnapshotForMaster(&MasterSnapshot->SnapShots,p)
| ;
43437|     }
43438|     return p;
43439| }
43440|
43441| /*-----
| -----*/
43442| /*
43443|     This routine can be called with it being exclusive
| or shared.
43444| */
43445| void DoneWithSnapshot( pkSnapshotEntry Snapshot )
43446| {
43447|     InterlockedDecrement((PLONG)&Snapshot->Count);
43448|

```

```

43449|  if((SnapShot->Deleted) && (SnapShot->Count==0)) {
43450|      BOOLEAN IsExclusive =
43451|      | IsSnapShotAcquiredForWrite();
43452|      ULONG IsShared = FALSE;
43453|      // looks like if IsSnapShotAcquiredForWrite
43454|      | returns true, then
43455|      // so will IsSnapShotAcquiredForRead, which is
43456|      | NOT what we want
43457|      // so only check shared access if not exclusive
43458|      if(!IsExclusive) {
43459|          IsShared = IsSnapShotAcquiredForRead();
43460|      }
43461|      // Cant be both at once!!!
43462| #ifdef DEBUG
43463|      if((IsExclusive) && (IsShared)) {
43464|          Debug(DEBUG_DEVSUP,("DoneWithSnapShot:
43465|      | Exclusive and shared??? tell me how please!!!\n"));
43466|          DbgBreakPoint();
43467|      } else
43468|      if((!IsExclusive) && (!IsShared)) {
43469|          Debug(DEBUG_DEVSUP,("DoneWithSnapShot: No
43470|      | resource held!!!!\n"));
43471|          DbgBreakPoint();
43472|      } else
43473|      if(IsExclusive) {
43474|          Debug(DEBUG_DEVSUP,("DoneWithSnapShot:
43475|      | Resource held exclusive!!!!\n"));
43476|      } else {
43477|          Debug(DEBUG_DEVSUP,("DoneWithSnapShot:
43478|      | Resource held shared!!!!\n"));
43479|      }
43480| #endif
43481|      if(IsShared) {
43482|          ReleaseSnapShotForRead();
43483|      }
43484|      // if not already exclusive then grab it.
43485|      if(!IsExclusive) {
43486|          GetSnapShotForWrite();
43487|      }
43488|      __try {
43489|          // free snapshot
43490|          Debug(DEBUG_DEVSUP,("DoneWithSnapShot:
43491|      | Snapshot %08x is deleted\n",SnapShot));
43492|      }
43493|      // remove resource or we will bug check the
43494|      | system

```

```

43490|         Dictionary::Destroy(SnapShot->Dictionary);
43491|
43492|         // zero out everything incase someone uses
         | them
43493|         RtlZeroMemory(SnapShot,sizeof(*SnapShot));
43494|
43495|         MemFreePool(SnapShot);
43496|         SnapShot=NULL;
43497|     } __finally {
43498|         if(!IsExclusive) {
43499|             // release it
43500|             ReleaseSnapShotForWrite();
43501|             // and go back to being shared if it
         | was
43502|             if(IsShared) {
43503|                 GetSnapShotForRead();
43504|             }
43505|         }
43506|     }
43507| } else {
43508|     if(SnapShot->Deleted) {
43509|         Debug(DEBUG_DEVSUP,("DoneWithSnapShot:
         | Snapshot %08x is deleted and still in use %08x
         | time(s)\n",SnapShot,SnapShot->Count));
43510|     }
43511| }
43512| return;
43513| }
43514|
43515| void UseSnapShot( pkSnapShotEntry SnapShot )
43516| {
43517|     InterlockedIncrement((PLONG)&SnapShot->Count);
43518|     return;
43519| }
43520|
43521| NTSTATUS
43522| PSManIrpCompletion(
43523|     IN PDEVICE_OBJECT DeviceObject,
43524|     IN PIRP Irp,
43525|     IN PVOID Context
43526| )
43527|
43528| /*++
43529|
43530| Routine Description:
43531|
43532|     Forwarded IRP completion routine. Set an event and
         | return
43533|     STATUS_MORE_PROCESSING_REQUIRED. Irp forwarder will
         | wait on this

```

```

43534|    event and then re-complete the irp after cleaning
      | up.
43535|
43536| Arguments:
43537|
43538|    DeviceObject is the device object of the WMI driver
43539|    Irp is the WMI irp that was just completed
43540|    Context is a PKEVENT that forwarder will wait on
43541|
43542| Return Value:
43543|
43544|    STATUS_MORE_PROCESSING_REQUIRED
43545|
43546| --*/
43547|
43548| {
43549|    PKEVENT Event = (PKEVENT) Context;
43550|
43551|    UNREFERENCED_PARAMETER(DeviceObject);
43552|    UNREFERENCED_PARAMETER(Irp);
43553|
43554|    pmSetEvent(Event);
43555|
43556|    return(STATUS_MORE_PROCESSING_REQUIRED);
43557|
43558| } // end PManIrpCompletion()
43559|
43560|
43561| NTSTATUS
43562| PManForwardIrpSynchronous(
43563|     IN PDEVICE_OBJECT DeviceObject,
43564|     IN PIRP Irp
43565| )
43566|
43567| /*++
43568|
43569| Routine Description:
43570|
43571|    This routine sends the Irp to the next driver in
      | line
43572|    when the Irp needs to be processed by the lower
      | drivers
43573|    prior to being processed by this one.
43574|
43575| Arguments:
43576|
43577|    DeviceObject
43578|    Irp
43579|
43580| Return Value:

```

```

43581|
43582|     NTSTATUS
43583|
43584|  --*/
43585|
43586|  {
43587|      KEVENT event;
43588|      NTSTATUS status;
43589|
43590|      PDEVICE_OBJECT Target=NULL;
43591|
43592|      switch(PsmGetObjectType(DeviceObject)) {
43593|          case OBJECT_FILTEREDDISK: {
43594|              PFILTERED_EXTENSION
43595|              | DevExt=GetFilteredExtension(DeviceObject);
43596|              Target = DevExt->TargetDeviceObject;
43597|              break;
43598|          }
43599|          case OBJECT_FS_FILTER: {
43600|              PFS_FILTER_EXTENSION DevExt =
43601|              | (PFS_FILTER_EXTENSION)
43602|              | GetDeviceExtension(DeviceObject);
43603|              Target = DevExt->TargetDeviceObject;
43604|              break;
43605|          }
43606|          default:
43607|              Debug(DEBUG_DEVSUP,("SendSync: Invalid request
43608|              | for object type %d\n",PsmGetObjectType(DeviceObject)));
43609|              ASSERT(FALSE);
43610|              return STATUS_INVALID_DEVICE_REQUEST;
43611|          }
43612|
43613|      ASSERT(Target != NULL);
43614|
43615|      KeInitializeEvent(&event, NotificationEvent,
43616|          | FALSE);
43617|
43618|      //
43619|      // copy the irpstack for the next device
43620|      //
43621|
43622|      IoCopyCurrentIrpStackLocationToNext(Irp);
43623|
43624|      //
43625|      // set a completion routine
43626|      //
43627|
43628|      IoSetCompletionRoutine(Irp, PSMAN_IrpCompletion,
43629|          &event, TRUE, TRUE, TRUE);
43630|
43631|

```

```

43626| //
43627| // call the next lower device
43628| //
43629|
43630| status = IoCallDriver(Target, Irp);
43631|
43632| //
43633| // wait for the actual completion
43634| //
43635|
43636| if (status == STATUS_PENDING) {
43637|     pmWaitForSingleObject(&event, NULL);
43638|     status = Irp->IoStatus.Status;
43639| }
43640|
43641| return status;
43642|
43643| } // end PManForwardIrpSynchronous()
43644|
43645| #define FILTER_DEVICE_PROPOGATE_FLAGS      0
43646| #define FILTER_DEVICE_PROPOGATE_CHARACTERISTICS
43647|     | (FILE_REMOVABLE_MEDIA | \
43648|     | FILE_READ_ONLY_DEVICE | \
43649|     | FILE_FLOPPY_DISKETTE | \
43650|     )
43651| VOID
43652| PManSyncFilterWithTarget(
43653|     IN PDEVICE_OBJECT FilterDevice,
43654|     IN PDEVICE_OBJECT TargetDevice
43655| )
43656| {
43657|     ULONG propFlags;
43658|
43659|     PAGED_CODE();
43660|
43661|     //
43662|     // Propagate all useful flags from target to psman.
43663|     // MountMgr will look
43664|     // at the psman object capabilities to figure out
43665|     // if the disk is
43666|     // a removable and perhaps other things.
43667|     //
43668|     propFlags = TargetDevice->Flags &
43669|         | FILTER_DEVICE_PROPOGATE_FLAGS;
43670|     FilterDevice->Flags |= propFlags;
43671|
43672|     propFlags = TargetDevice->Characteristics &

```

```

    | FILTER_DEVICE_PROPOGATE_CHARACTERISTICS;
43670|   FilterDevice->Characteristics |= propFlags;
43671| }
43672|
43673|
43674| VOID
43675| PSMANAddCounters(
43676|   IN OUT PDISK_PERFORMANCE TotalCounters,
43677|   IN PDISK_PERFORMANCE NewCounters
43678|   )
43679| {
43680|   TotalCounters->BytesRead.QuadPart +=
    | NewCounters->BytesRead.QuadPart;
43681|   TotalCounters->BytesWritten.QuadPart+=
    | NewCounters->BytesWritten.QuadPart;
43682|   TotalCounters->ReadTime.QuadPart +=
    | NewCounters->ReadTime.QuadPart;
43683|   TotalCounters->WriteTime.QuadPart +=
    | NewCounters->WriteTime.QuadPart;
43684| #if _WIN32_WINNT>=0x0500
43685|   TotalCounters->IdleTime.QuadPart +=
    | NewCounters->IdleTime.QuadPart;
43686|   TotalCounters->ReadCount      +=
    | NewCounters->ReadCount;
43687|   TotalCounters->WriteCount     +=
    | NewCounters->WriteCount;
43688|   TotalCounters->SplitCount     +=
    | NewCounters->SplitCount;
43689| #endif
43690| }
43691|
43692|
43693| int CreateJunction( const PWCHAR LinkDirectory, const
    | PWCHAR LinkTarget, LARGE_INTEGER Time )
43694| {
43695|   UNICODE_STRING  UniName={0};
43696|   char    reparseBuffer[MAX_PATH*3];
43697|   OBJECT_ATTRIBUTES ObjectAttributes={0};
43698|   IO_STATUS_BLOCK IoStatus;
43699|   WCHAR    targetNativeFileName[MAX_PATH];
43700|   HANDLE    FileHandle;
43701|   NTSTATUS Status;
43702|   PREPARSE_MOUNTPOINT_DATA_BUFFER reparseInfo =
43703|   (PREPARSE_MOUNTPOINT_DATA_BUFFER)
    | reparseBuffer;
43704|
43705|
43706|   SbTouchVolume(LinkTarget);
43707|
43708|   wcscpy(targetNativeFileName,LinkTarget);

```



```

43709|   RtlInitUnicodeString( &UniName, LinkDirectory );
43710|
43711|   Debug(DEBUG_DICT,("Creating junction for '%S' to
    | '%S'\n",
43712|       LinkDirectory,
43713|       targetNativeFileName));
43714|
43715|   //
43716|   // Create the directory to be linked
43717|   //
43718|
43719|   NTSTATUS createDirStatus = SbCreateDirectory (
    | LinkDirectory, NULL, FILE_ATTRIBUTE_NORMAL );
43720|   Debug(DEBUG_DICT,("CreateJunction:
    | SbCreateDirectory('%S') returned %08x\n",
43721|       LinkDirectory,
43722|       createDirStatus));
43723|
43724|   //
43725|   // Create the link
43726|   //
43727|   InitializeObjectAttributes ( &ObjectAttributes,
43728|                               &UniName,
43729|                               OBJ_CASE_INSENSITIVE,
43730|                               NULL,
43731|                               NULL );
43732|
43733|   Status = ZwCreateFile( &FileHandle,
43734|       GENERIC_WRITE,
    | // desired access
43735|       &ObjectAttributes,
    | // object attributes
43736|       &IoStatus,
43737|       NULL,          //
    | alloc size
43738|       FILE_ATTRIBUTE_NORMAL,
    | // file attributes
43739|       FILE_SHARE_WRITE |
    | FILE_SHARE_READ,          // share
    | access
43740|       FILE_OPEN,
    | // create disposition
43741|       /*FILE_DIRECTORY_FILE|*/
43742|       | FILE_OPEN_REPARSE_POINT|
43743|       | FILE_OPEN_FOR_BACKUP_INTENT,          // create
    | options
43744|       NULL, // eabuffer
43745|       0 ); // ealength

```

```

43746|
43747|     if(NT_SUCCESS(Status)) {
43748|         FILE_BASIC_INFORMATION Basic;
43749|
43750|         // set creation date and time
43751|         Status = ZwQueryInformationFile(
43752|             FileHandle,           // IN HANDLE
43753|             | FileHandle,
43754|             &IoStatus,           // OUT
43755|             | PIO_STATUS_BLOCK IoStatusBlock,
43756|             &Basic,              // IN PVOID
43757|             | FileInformation,
43758|             sizeof(Basic),       // IN ULONG
43759|             | Length,
43760|             FileBasicInformation// IN
43761|             | FILE_INFORMATION_CLASS FileInformationClass
43762|         );
43763|         Basic.CreationTime = Time;
43764|
43765|         Status = ZwSetInformationFile(
43766|             FileHandle,           // IN HANDLE
43767|             | FileHandle,
43768|             &IoStatus,           // OUT
43769|             | PIO_STATUS_BLOCK IoStatusBlock,
43770|             &Basic,              // IN PVOID
43771|             | FileInformation,
43772|             sizeof(Basic),       // IN ULONG
43773|             | Length,
43774|             FileBasicInformation // IN
43775|             | FILE_INFORMATION_CLASS FileInformationClass
43776|         );
43777|         //
43778|         // Build the reparse info
43779|         //
43780|         RtlZeroMemory( reparseInfo, sizeof(
43781|             | REPARSE_MOUNTPOINT_DATA_BUFFER ));
43782|         reparseInfo->ReparseTag =
43783|             | IO_REPARSE_TAG_MOUNT_POINT;
43784|
43785|         reparseInfo->ReparseTargetLength = wcslen(
43786|             | targetNativeFileName ) * sizeof(WCHAR);
43787|         reparseInfo->ReparseTargetMaximumLength =
43788|             | reparseInfo->ReparseTargetLength + sizeof(WCHAR);
43789|         wcscpy( reparseInfo->ReparseTarget,
43790|             | targetNativeFileName );
43791|         reparseInfo->ReparseDataLength =
43792|             | reparseInfo->ReparseTargetLength + 12;
43793|
43794|         Debug(DEBUG_DEVSUP,("rt=%08x, rdl=%08x, r=%04x,

```

```

    | rtl=%04x, rtml=%04x, r1=%04x, '%S'\n",
43780|         reparseInfo->ReparseTag,
43781|         reparseInfo->ReparseDataLength,
43782|         reparseInfo->Reserved,
43783|         reparseInfo->ReparseTargetLength,
43784|         reparseInfo->ReparseTargetMaximumLength,
43785|         reparseInfo->Reserved1,
43786|         reparseInfo->ReparseTarget
43787|     ));
43788|
43789|     //
43790|     // Set the link
43791|     //
43792|     Status = ZwFsControlFile(
43793|         FileHandle,
43794|         NULL,
43795|         NULL,
43796|         NULL,
43797|         &IoStatus,
43798|         FSCTL_SET_REPARSE_POINT,
43799|         reparseInfo,
43800|         reparseInfo->ReparseDataLength +
        | REPARSE_MOUNTPOINT_HEADER_SIZE,
43801|         NULL,
43802|         NULL
43803|     );
43804|
43805|     if(NT_SUCCESS(Status)) {
43806|         Debug(DEBUG_DICT,("Success creating link
        | '%S' to '%S'\n",LinkDirectory,LinkTarget));
43807|     } else {
43808|         Debug(DEBUG_DICT,("Error %08x sending
        | ioctl\n",Status));
43809|     }
43810|     ZwClose(FileHandle);
43811|
43812| } else {
43813|     Debug(DEBUG_DICT,("Error %08x creating junction
        | for '%S' to '%S'\n",Status,LinkDirectory,LinkTarget));
43814| }
43815|
43816| return 0;
43817| }
43818|
43819| /*
43820| Determines if a request is pending for 1 or more in
        | the
43821| range specified
43822| */
43823| ULONG IsBeingProcessed ( tWriteRequest *Request )

```

```

43824| {
43825|     LIST_ENTRY *ListEntry;
43826|     unsigned __int64 RSector1 =
        | Request->RoundedSector.QuadPart;
43827|     unsigned __int64 RSector2 =
        | Request->RoundedSector.QuadPart + Request->RoundedCount
        | - 1;
43828|     ULONG FoundOne=0;
43829|     KIRQL oldIrql;
43830|
43831|     pmAcquireSpinLock(&WriteSpinLock,&oldIrql);
43832|     ListEntry = ProcessingQueue.Flink;
43833|
43834|     while(ListEntry! =&ProcessingQueue) {
43835|         tWriteRequest *p =
            | CONTAINING_RECORD(ListEntry,tWriteRequest,ProcessingEntr
            | y);
43836|         unsigned __int64 PSector1 =
            | p->RoundedSector.QuadPart;
43837|         unsigned __int64 PSector2 =
            | p->RoundedSector.QuadPart + p->RoundedCount - 1;
43838|
43839|         if( (p->DeviceObject == Request->DeviceObject))
            | {
43840| /*
43841|             Sector Count  Sector Count
43842|             0          1
            | 0          1
43843|             0          1
            | 0          8
43844|             1          1
            | 0          8
43845|             8          1
            | 0          8
43846|             0          8
            | 0          8
43847|             0          8
            | 0          1
43848|             0          8
            | 1          3
43849|             0          2
            | 1          3
43850|             0          2
            | 1          1
43851|             2          1
            | 1          3
43852| */
43853|
43854|             if ( (RSector1>=PSector1 &&
                | RSector1<=PSector2) ||

```

```

43855|          (RSector2>=PSector1 &&
| RSector2<=PSector2) ||
43856|          (PSector1>=RSector1 &&
| PSector1<=RSector2) ||
43857|          (PSector2>=RSector1 &&
| PSector2<=RSector2) ) {
43858|          FoundOne = 1;
43859|          break;
43860|      }
43861|  }
43862|
43863|  ListEntry = ListEntry->Flink;
43864|  }
43865|
43866|  pmReleaseSpinLock(&WriteSpinLock,oldIrql);
43867|  return FoundOne;
43868| }
43869|
43870| /*
43871|  Determines if a request is pending for 1 or more in
| the
43872|  range specified, if it is not then add request to
| queue
43873| */
43874| ULONG IsBeingProcessedEx ( tWriteRequest *Request )
43875| {
43876|  LIST_ENTRY *ListEntry;
43877|  unsigned __int64 RSector1 =
| Request->RoundedSector.QuadPart;
43878|  unsigned __int64 RSector2 =
| Request->RoundedSector.QuadPart + Request->RoundedCount
| - 1;
43879|  ULONG FoundOne=0;
43880|  KIRQL oldIrql;
43881|
43882|  pmAcquireSpinLock(&WriteSpinLock,&oldIrql);
43883|  ListEntry = ProcessingQueue.Flink;
43884|
43885|  while(ListEntry! =&ProcessingQueue) {
43886|      tWriteRequest *p =
| CONTAINING_RECORD(ListEntry,tWriteRequest,ProcessingEntr
| y);
43887|      unsigned __int64 PSector1 =
| p->RoundedSector.QuadPart;
43888|      unsigned __int64 PSector2 =
| p->RoundedSector.QuadPart + p->RoundedCount - 1;
43889|
43890|      if( (p->DeviceObject == Request->DeviceObject))
| {
43891|  /*

```

```

43892|          Sector Count Sector Count
43893|          0          1
43894|          | 0          1
43895|          | 0          8
43896|          | 0          8
43897|          | 0          8
43898|          | 0          8
43899|          | 0          8
43900|          | 1          3
43901|          | 1          3
43902|          | 1          1
43903|          | 1          3
43904|          | 1          3
43905|          | 1          3
43906|          | 1          3
43907|          | 1          3
43908|          | 1          3
43909|          | 1          3
43910|          | 1          3
43911|          | 1          3
43912|          | 1          3
43913|          | 1          3
43914|          | 1          3
43915|          | 1          3
43916|          | 1          3
43917|          | 1          3
43918|          | 1          3
43919|          | 1          3
43920|          | 1          3
43921|          | 1          3
43922|          | 1          3
43923|          | 1          3
43924|          | 1          3
43925|          | 1          3

```

```

43926|
43927| NTSTATUS FS_GetVolumeBitmap( PFILE_OBJECT FileObject,
    | STARTING_LCN_INPUT_BUFFER *SLIB, VOLUME_BITMAP_BUFFER
    | *VB, ULONG VBSize )
43928| {
43929|     PIRP                Irp=NULL;
43930|     KEVENT               Event={0};
43931|     IO_STATUS_BLOCK       IoStatusBlock={0};
43932|     NTSTATUS              Status=0;
43933|     PIO_STACK_LOCATION    IrpStack=NULL;
43934|     PDEVICE_OBJECT        DeviceObject;
43935|     ULONG                 Ret = 0;
43936|
43937|     __try {
43938|
43939|         DeviceObject =
            | IoGetRelatedDeviceObject(FileObject);
43940|
43941|         //
43942|         // Set the event object to the unsigned
            | state.
43943|         // It will be used to signal request
            | completion.
43944|         //
43945|
43946|         KeInitializeEvent(&Event, NotificationEvent,
            | FALSE);
43947|
43948|
43949|         //
43950|         // Create IRP for read
43951|         //
43952|
43953|         Irp = IrpAllocateIrp(DeviceObject->StackSize);
43954|
43955|         if (!Irp) {
43956|             Debug(DEBUG_DEVSUP,("FS_GetVolumeBitmap:
            | Error! Unable to allocate irp\n"));
43957|             try_return(Status =
            | STATUS_INSUFFICIENT_RESOURCES);
43958|         }
43959|
43960|         Irp->Flags = 0; //IRP_READ_OPERATION;
43961|
43962|         Irp->AssociatedIrp.SystemBuffer = NULL;
43963|         Irp->MdlAddress = NULL;
43964|         Irp->UserBuffer = VB;
43965|         Irp->UserEvent = &Event;
43966|         Irp->UserIoStb = &IoStatusBlock;
43967|         /*lint -save -e740 */

```

```

43968|     Irp->Tail.Overlay.Thread =
      | PsGetCurrentThread();
43969|     /*lint -restore */
43970|     Irp->Tail.Overlay.OriginalFileObject = NULL;
43971|     Irp->RequestorMode =
      | (KPROCESSOR_MODE)KernelMode;
43972|     Irp->IoStatus.Status          =
      | STATUS_PENDING;
43973|     Irp->IoStatus.Information      = 0;
43974|
43975|     IrpStack = IoGetNextIrpStackLocation(Irp);
43976|     RtlZeroMemory((PVOID)IrpStack,
      | sizeof(IO_STACK_LOCATION));
43977|     // just sets event
43978|     IoSetCompletionRoutine(Irp,
      | Sblo_ReadDeviceMdlCompletionRoutine, NULL, TRUE, TRUE,
      | TRUE);
43979|     IrpStack->MajorFunction =
      | IRP_MJ_FILE_SYSTEM_CONTROL;
43980|     IrpStack->MinorFunction =
      | IRP_MN_USER_FS_REQUEST;
43981|
      | IrpStack->Parameters.DeviceIoControl.IoControlCode =
      | FSCTL_GET_VOLUME_BITMAP;
43982|
      | IrpStack->Parameters.DeviceIoControl.OutputBufferLength
      | = VBSIZE;
43983|
      | IrpStack->Parameters.DeviceIoControl.InputBufferLength
      | = sizeof(STARTING_LCN_INPUT_BUFFER);
43984|
      | IrpStack->Parameters.DeviceIoControl.Type3InputBuffer =
      | SLIB;
43985|     IrpStack->DeviceObject =
      | FileObject->DeviceObject;
43986|     IrpStack->FileObject = FileObject;
43987|
43988|     Status = IoCallDriver(DeviceObject, Irp);
43989|
43990|     if (Status == STATUS_PENDING) {
43991|
43992|         Debug(DEBUG_DEVSUP,("FS_GetVolumeBitMap:
      | Waiting for get bitmap to finish\n"));
43993|         ASSERT(KeGetCurrentIrql() <
      | DISPATCH_LEVEL);
43994|         pmWaitForSingleObject(&Event,NULL);
43995|
43996|         Status = IoStatusBlock.Status;
43997|         Ret = Irp->IoStatus.Information;
43998|     }

```



```

43999|     IrpFreeIrp(Irp);
44000|
44001| #if DO_ALL_IO
44002|     Debug(DEBUG_DEVSUP,("FS_GetVolumeBitmap
    | Status=%08x, size=%08x (%x,%x), Wanted=%l64x,
    | got=%l64x, left=%l64x\n",
44003|         Status,
44004|         VBSize,
44005|         IoStatusBlock.Information,
44006|         Ret,
44007|         SLIB->StartingLcn,
44008|         VB->StartingLcn,
44009|         VB->BitmapSize
44010|     ));
44011| #endif
44012|
44013| try_exit: NOTHING;
44014| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
44015|     Status = GetExceptionCode();
44016|     Debug(DEBUG_DEVSUP,("FS_GetVolumeBitMap:
    | Exception %08x\n",Status));
44017| }
44018|
44019| return Status;
44020| }
44021|
44022| unsigned __int64 GetVolumeBitmapSize( PFILE_OBJECT
    | VolumeObject )
44023| {
44024|     STARTING_LCN_INPUT_BUFFER SLIB={0};
44025|     VOLUME_BITMAP_BUFFER VB={0};
44026|     NTSTATUS Status = FS_GetVolumeBitmap( VolumeObject,
    | &SLIB, &VB, sizeof(VOLUME_BITMAP_BUFFER));
44027|
44028|     if(Status==STATUS_BUFFER_OVERFLOW) {
44029|         // dword align buffer size
44030|         return (DWORD_ALIGN(VB.BitmapSize.QuadPart) /
    | 8)+FIELD_OFFSET(VOLUME_BITMAP_BUFFER, Buffer);
44031|     } else {
44032|         Debug(DEBUG_DEVSUP,("GetVolumeBitMapSize: Error
    | %08x\n",Status));
44033|         ASSERT(Status==STATUS_SUCCESS);
44034|         return 0;
44035|     }
44036| }
44037|
44038|
44039| //-----
    | -----

```

```

44040| #define BYTES_PER_WORK_UNIT    (4096)
44041| #define CLUSTERS_PER_WORK_UNIT  (8*BYTES_PER_WORK_UNIT)
44042| // Gets and coalesces the volume bit map as of the
    | snapshot into a GranuleBitMap. If no map already exists
    | it becomes the
44043| // UsedGranuleBitMap as all used need to be marked for
    | caching, otherwise it is used to reset all used
    | granules as needing
44044| // to be recached for this snapshot.
44045|
44046| // NOTE: Since we may be dealing with large bitmaps
    | which could take significant time to read this has to
    | occur
44047| //      asynchronously while writes may be being
    | processed. All writes have to be cached during this
    | time.
44048| //      Sure, it may be superfluous! but it will be
    | SAFE.
44049|
44050|
44051| ULONG ReassessUsedGranuleBitMap (
44052|     PFILE_OBJECT  VolumeObject,
44053|     LARGE_INTEGER  NumClustersInVolume,
44054|     ULONG          ClusterSizeInBytes,
44055|     PRTL_BITMAP    *GranuleBitMap )
44056| {
44057|     ULONG status = STATUS_SUCCESS;
44058|     ULONG ClustersPerGranule = GRANULE_SIZE /
    | ClusterSizeInBytes;
44059|     ULONG NumGranules =
    | (ULONG)((NumClustersInVolume.QuadPart +
    | (ClustersPerGranule-1)) / ClustersPerGranule);
44060|     ULONG GranuleIndex=0;
44061|     LARGE_INTEGER ClusterIndex={0};
44062|     ULONG k=0;
44063|     ULONG GranuleBitMapSizeInBytes =
    | (sizeof(RTL_BITMAP)+(NumGranules+31) / 32) * 4);
44064|     RTL_BITMAP WorkUnitBitMap;
44065|     // Fastfat.sys seems to be off by one byte when
    | calculating the amount of data to transfer
44066|     // into our buffer. we dont care about the end, as
    | we will reread at starting at that byte anyway
44067|     VOLUME_BITMAP_BUFFER *WorkUnit =
    | (VOLUME_BITMAP_BUFFER *)
    | MemAllocatePoolWithTag(PagedPool,FIELD_OFFSET(VOLUME_BIT
    | MAP_BUFFER,Buffer))+
    | BYTES_PER_WORK_UNIT+sizeof(DWORD),TEMPTAG );
44068|
44069|     ASSERT ( GRANULE_SIZE >= ClusterSizeInBytes );
44070|     ASSERT ( GRANULE_SIZE % ClusterSizeInBytes == 0 );

```

```

44071| // we can only handle 32bits for granule size
44072| ASSERT ( ((NumClustersInVolume.QuadPart +
| (ClustersPerGranule-1)) / ClustersPerGranule) <
| 0x00000000ffffffffUI64);
44073|
44074| if (!(*GranuleBitMap)) {
44075|     *GranuleBitMap = (PRTL_BITMAP)
| MemAllocatePoolWithTag(
| PagedPool,GranuleBitMapSizeInBytes,PSM_FREE_SPACE_TAG
| );
44076|     if (*GranuleBitMap) {
44077|         RtlInitializeBitMap(*GranuleBitMap,(PULONG)(*GranuleBitM
| ap + 1),NumGranules );
44078|         RtlClearAllBits ( *GranuleBitMap );
44079|     }
44080| }
44081| if (!(*GranuleBitMap) || !WorkUnit ) {
44082|     Debug( DEBUG_DEVSUP,( "Out of memory in
| BuildUsedGranuleBitMap\n" ));
44083|     status = STATUS_INSUFFICIENT_RESOURCES;
44084| } else {
44085|     RtlZeroMemory( WorkUnit,
| FIELD_OFFSET(VOLUME_BITMAP_BUFFER,Buffer)+
| BYTES_PER_WORK_UNIT); // not really needed
44086|     RtlInitializeBitMap(&WorkUnitBitMap,(PULONG>(&WorkUnit->
| Buffer),CLUSTERS_PER_WORK_UNIT);
44087|
44088|     for ( GranuleIndex=0; GranuleIndex <
| NumGranules; ++GranuleIndex ) {
44089|         BOOLEAN GranuleIsUsed = FALSE;
44090|         ClusterIndex.QuadPart = (unsigned
| __int64)ClustersPerGranule * GranuleIndex;
44091|
44092|         for ( k=0; k<ClustersPerGranule; ++k,
| ++ClusterIndex.QuadPart ) {
44093|             // calculate the index of the bit
| within the work unit...
44094|             ULONG workUnitBitOffset =
| (ULONG)(ClusterIndex.QuadPart %
| CLUSTERS_PER_WORK_UNIT);
44095|             if ( workUnitBitOffset == 0 ) {
44096|
44097|                 // time to slurp another batch of
| bits
44098|                 ULONGLONG
| BitsToFetch=NumClustersInVolume.QuadPart -
| ClusterIndex.QuadPart;
44099|                 ULONG bytesToFetch =

```

```

    | (ULONG)((BitsToFetch+7) / 8);
44100|
44101|         if ( bytesToFetch >
    | BYTES_PER_WORK_UNIT ) {
44102|         bytesToFetch =
    | BYTES_PER_WORK_UNIT;
44103|     }
44104|     bytesToFetch +=
    | FIELD_OFFSET(VOLUME_BITMAP_BUFFER,Buffer);
44105|
44106|     STARTING_LCN_INPUT_BUFFER SLIB;
44107|     SLIB.StartingLcn = ClusterIndex;
44108|
44109|     status = FS_GetVolumeBitmap(
    | VolumeObject, &SLIB, WorkUnit, bytesToFetch );
44110|
44111|     if(status==STATUS_BUFFER_OVERFLOW)
    | {
44112|         // this is not an error, it is
    | just a warning saying we didnt get all
44113|         // the data that is left
44114|         status = STATUS_SUCCESS;
44115|     }
44116|
44117|     if ( !INT_SUCCESS(status)) {
44118|         Debug( DEBUG_DEVSUP,( "Error
    | retrieving work unit in BuildUsedGranuleBitMap\n" ));
44119|         goto cleanup;
44120|     }
44121|
44122|     if ( GranuleIndex==0) {
44123|         // Only after getting the very
    | first chunk of virtual volume bitmap do we need to...
44124|
44125|         // do a sanity check that the
    | virtual volume is also at ground zero
44126|         ASSERT
    | (WorkUnit->StartingLcn.QuadPart==0);
44127|
44128|         // ... and in case the virtual
    | volume is smaller than the live ( because of subsequent
    | dynamic volume expansion)
44129|         // use the size of the virtual
    | bitmap we've found as the size we're going to examine
    | in this reassessment call
44130|
44131|         NumClustersInVolume.QuadPart =
    | WorkUnit->BitmapSize.QuadPart;
44132|         NumGranules =
    | (ULONG)((NumClustersInVolume.QuadPart +

```

```

    | (ClustersPerGranule-1)) / ClustersPerGranule);
44133|         Debug( DEBUG_DEVSUP,(
    | "Restricting reassessment to Virtual volumes smaller
    | bitmap size (%l64d = 0x%l64x)\n"
44134|         ,WorkUnit->BitmapSize
44135|         ,WorkUnit->BitmapSize
44136|     ));
44137|     }
44138|
44139| #if DO_ALL_BITMAPS
44140|         //FIXFIXFIX need an assert to make
    | sure workunit fits 32bits
44141|         for (ULONG i= 0;
    | i*4<bytesToFetch-FIELD_OFFSET(VOLUME_BITMAP_BUFFER,Buffer
    | r); i+=0x8 ) {
44142|             if ( (i==0) ||
44143|                 (
    | ((PULONG)(WorkUnit->Buffer))[i+0] !=
    | ((PULONG)(WorkUnit->Buffer))[i-8+0] ) ||
44144|                 (
    | ((PULONG)(WorkUnit->Buffer))[i+1] !=
    | ((PULONG)(WorkUnit->Buffer))[i-8+1] ) ||
44145|                 (
    | ((PULONG)(WorkUnit->Buffer))[i+2] !=
    | ((PULONG)(WorkUnit->Buffer))[i-8+2] ) ||
44146|                 (
    | ((PULONG)(WorkUnit->Buffer))[i+3] !=
    | ((PULONG)(WorkUnit->Buffer))[i-8+3] ) ||
44147|                 (
    | ((PULONG)(WorkUnit->Buffer))[i+4] !=
    | ((PULONG)(WorkUnit->Buffer))[i-8+4] ) ||
44148|                 (
    | ((PULONG)(WorkUnit->Buffer))[i+5] !=
    | ((PULONG)(WorkUnit->Buffer))[i-8+5] ) ||
44149|                 (
    | ((PULONG)(WorkUnit->Buffer))[i+6] !=
    | ((PULONG)(WorkUnit->Buffer))[i-8+6] ) ||
44150|                 (
    | ((PULONG)(WorkUnit->Buffer))[i+7] !=
    | ((PULONG)(WorkUnit->Buffer))[i-8+7] ) ) {
44151|
44152|         Debug(DEBUG_DEVSUP,("
    | Granule/Cluster: %8l64x/%08l64x: %08x %08x %08x %08x
    | - %08x %08x %08x %08x\n"
44153|         | ,(WorkUnit->StartingLcn.QuadPart+i*0x20)/ClustersPerGran
    | ule
44154|         | ,(WorkUnit->StartingLcn.QuadPart+i*0x20)
44155|

```

```

    |,((PULONG)(WorkUnit->Buffer))[i+0x0]
44156|
    |,((PULONG)(WorkUnit->Buffer))[i+0x1]
44157|
    |,((PULONG)(WorkUnit->Buffer))[i+0x2]
44158|
    |,((PULONG)(WorkUnit->Buffer))[i+0x3]
44159|
    |,((PULONG)(WorkUnit->Buffer))[i+0x4]
44160|
    |,((PULONG)(WorkUnit->Buffer))[i+0x5]
44161|
    |,((PULONG)(WorkUnit->Buffer))[i+0x6]
44162|
    |,((PULONG)(WorkUnit->Buffer))[i+0x7]
44163|                ));
44164|                }
44165|                }
44166| #endif
44167|                }
44168|
44169|                if ( ClusterIndex.QuadPart >=
    | NumClustersInVolume.QuadPart ) {
44170|                break; // this is why
    | memset(WorkUnit,...) above not needed.
44171|                }
44172|
44173|                PsmBitPositionValidate
    | (&WorkUnitBitMap, workUnitBitOffset);
44174|
    | if(RtlCheckBit(&WorkUnitBitMap,workUnitBitOffset)) {
44175|                // If one cluster is used, the
    | whole granule is used...
44176|                GranuleIsUsed = TRUE;
44177|                break; // don't waste time looking
    | at other clusters in this granule
44178|                }
44179|                }
44180|
44181|                if ( GranuleIsUsed ) {
44182|                PsmBitPositionValidate (*GranuleBitMap,
    | GranuleIndex);
44183|                RtlSetBits ( *GranuleBitMap,
    | GranuleIndex, 1 );
44184|                }
44185|                }
44186|                }
44187|
44188| cleanup:
44189|    if ( !NT_SUCCESS(status) ) {

```

```

44190|     if ( *GranuleBitMap ) {
44191|         MemFreePool( *GranuleBitMap );
44192|         *GranuleBitMap = NULL;
44193|     }
44194| }
44195|
44196| if ( WorkUnit ) {
44197|     MemFreePool( WorkUnit );
44198|     WorkUnit = NULL;
44199| }
44200|
44201| return status;
44202| }
44203|
44204| /*
44205|     NewSize is the size in <<Sectors>> to extend to.
44206| */
44207| NTSTATUS ExtendFreeSpaceBitmaps( PDEVICE_OBJECT Volume,
    | LARGE_INTEGER NewSize )
44208| {
44209|     NTSTATUS Status = STATUS_UNSUCCESSFUL;
44210|     pDictionary po=NULL;
44211|     PFILTERED_EXTENSION
    | DevExt=GetFilteredExtension(Volume);
44212|
44213|     __try {
44214|         if((PersistentDictionary::DoFreeSpaceChecks())
    | && (DevExt->PSMed)) {
44215|
    | PersistentDictionary::GetDictionaryForVolume(Volume,po);
44216|         if ( po ) {
44217|             PRTL_BITMAP OldMap=NULL;
44218|             ULONG InTransform=0;
44219|             pPersistentDictionary
    | p=(pPersistentDictionary)po;
44220|             ULONG ClusterSize =
    | p->GetVolumeClusterSize();
44221|
44222|             // Cluster size cant be zero if
    | DoFreeSpaceChecks is on
44223|             ASSERT(ClusterSize);
44224|             ASSERT(GRANULE_SIZE>=ClusterSize);
44225|
44226|             ULONG ClustersPerGranule = GRANULE_SIZE
    | / ClusterSize;
44227|             ULONGLONG NewSizeInClusters =
    | ((NewSize.QuadPart*512)+(ClusterSize-1)) / ClusterSize;
44228|             ULONG NumGranules =
    | (ULONG)((NewSizeInClusters + (ClustersPerGranule-1)) /
    | ClustersPerGranule);

```

```

44229|         ULONG GranuleBitMapSizeInBytes =
      | (sizeof(RTL_BITMAP)+(NumGranules+31) / 32) * 4);
44230|
44231|         // allocate memory before getting the
      | snapshot for write access
44232|         // as we are allocating paged memory,
      | and io may need to occur
44233|         PRTL_BITMAP NewMap = (PRTL_BITMAP)
      | MemAllocatePoolWithTag(
      | PagedPool,GranuleBitMapSizeInBytes,PSM_FREE_SPACE_TAG
      | );
44234|
44235|         if (NewMap) {
44236|
44237|             | RtlInitializeBitMap(NewMap,(PULONG)(NewMap+
      | 1),NumGranules );
44238|
44239|             // lock out everybody from
      | accessing the free space map.
44240|             GetSnapShotForWrite();
44241|
44242|             __try {
44243|                 OldMap =
      | p->GetVolumeCachingMap(0);
44244|                 if(!OldMap) {
44245|                     OldMap =
      | p->GetVolumeCachingMap(1);
44246|                     InTransform = 1;
44247|                 }
44248|
44249|                 if(OldMap) {
44250|                     // make sure we are
      | actually increasing and not decreasing
44251|                     | if(NewMap->SizeOfBitMap>OldMap->SizeOfBitMap) {
44252|
44253|                         RtlMoveMemory(
44254|                             NewMap->Buffer,
44255|                             OldMap->Buffer,
44256|                             | (OldMap->SizeOfBitMap+7) / 8);
44257|
44258|                     // clear out the new
      | extended area
44259|                     PsmBitRangeValidate
      | (NewMap, OldMap->SizeOfBitMap, NewMap->SizeOfBitMap -
      | OldMap->SizeOfBitMap);
44260|                     | RtlClearBits(NewMap,OldMap->SizeOfBitMap,NewMap->SizeOfB

```



```

    | itMap-OldMap->SizeOfBitMap);
44261|
44262|
    | p->SetVolumeCachingMap(InTransform,NewMap);
44263|
44264|             MemFreePool(OldMap);
44265|
44266|             Status =
    | STATUS_SUCCESS;
44267|             } else {
44268|                 // Told to shrink.
44269|                 Status =
    | STATUS_SUCCESS;
44270|                 MemFreePool(NewMap);
44271|                 // this is to just
    | catch this weird case
44272|                 ASSERT(FALSE);
44273|             }
44274|             } else {
44275|                 // if this happens, then
    | the free space logic hasnt started yet for
44276|                 // the very first snapshot,
    | when it does start, it will be using the new size
44277|                 Status = STATUS_SUCCESS;
44278|                 MemFreePool(NewMap);
44279|             }
44280|             } __finally {
44281|                 ReleaseSnapShotForWrite();
44282|             }
44283|             } else {
44284|                 Debug(DEBUG_DICT,("Insufficient
    | memory to extend granule bitmap for volume
    | %08x\n",Volume));
44285|                 Status =
    | STATUS_INSUFFICIENT_RESOURCES;
44286|                 // we havent coded this condition
    | yet FIXFIXFIX
44287|                 ASSERT(FALSE);
44288|             }
44289|             } else {
44290|                 Debug(DEBUG_DICT,("No dictionary for
    | volume %08x\n",Volume));
44291|                 Status = STATUS_NO_SUCH_DEVICE;
44292|             }
44293|             } else {
44294|                 // nothing to update
44295|                 Status = STATUS_SUCCESS;
44296|             }
44297|     } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {

```

```

44298|     Status = GetExceptionCode();
44299|     Debug(DEBUG_DICT,("Exception %08x in
    | ExtendFreeSpaceBitmaps\n",Status));
44300| }
44301|
44302| return Status;
44303| }
44304|
44305|
44306| NTSTATUS FindAndProcessVirtualVolumeBitMap( const WCHAR
    | *VirtualVolName, const WCHAR *LiveVolName, PRTL_BITMAP
    | *BitMap, ULONG &ClusterSize )
44307| {
44308| #if 1
44309|     UNICODE_STRING  UniName={0};
44310|     WCHAR *Buffer=(WCHAR*)MemAllocateString(256);
44311|     OBJECT_ATTRIBUTES ObjectAttributes={0};
44312|     IO_STATUS_BLOCK IoStatus;
44313|     HANDLE  FileHandle;
44314|     PFILE_OBJECT  LiveFileObject;
44315|     PFILE_OBJECT  VirtualFileObject;
44316|     NTSTATUS Status;
44317|
44318|     if(!Buffer) {
44319|         return STATUS_INSUFFICIENT_RESOURCES;
44320|     }
44321|
44322| #ifdef DEBUG
44323|     if(IsSnapShotAcquiredForWrite()) {
44324|         // the reason this is bad is that we have the
            | writer lock
44325|         // and any io that needs to be PSMed, needs to
            | acquire the reader lock
44326|         // thus producing a deadlock, to fix it, dont
            | call with writer lock
44327|         // which you can do by spinning off whatever
            | you are trying to do to a
44328|         // worker thread.
44329|
            | Debug(DEBUG_DCPSM,("FindAndProcessVirtualVolumeBitMap:
            | Snapshot resource acquired for write!\n"));
44330|         DbgBreakPoint();
44331|     }
44332| #endif
44333|
44334|     SbTouchVolume(VirtualVolName);
44335|
44336|     wcscpy(Buffer,VirtualVolName);
44337|     RtlInitUnicodeString(&UniName,Buffer);
44338|

```

```

44339| InitializeObjectAttributes ( &ObjectAttributes,
44340|                               &UniName,
44341|                               OBJ_CASE_INSENSITIVE,
44342|                               NULL,
44343|                               NULL );
44344|
44345| Status = ZwCreateFile( &FileHandle,
44346|                       GENERIC_READ,
44347|                       | // desired access
44348|                       &ObjectAttributes,
44349|                       | // object attributes
44350|                       &IoStatus,
44351|                       NULL,          //
44352|                       | alloc size
44353|                       FILE_ATTRIBUTE_NORMAL,
44354|                       | // file attributes
44355|                       FILE_SHARE_WRITE |
44356|                       | FILE_SHARE_READ,          // share
44357|                       | access
44358|                       FILE_OPEN,
44359|                       | // create disposition
44360|                       0,          // create
44361|                       | options
44362|                       NULL, // eabuffer
44363|                       0 ); // ealength
44364| if(NT_SUCCESS(Status)) {
44365|     Status = ObReferenceObjectByHandle(
44366|         FileHandle,    // IN HANDLE Handle,
44367|         GENERIC_READ,
44368|         NULL,          // IN POBJECT_TYPE
44369|         | ObjectType,    // optional
44370|         (KPROCESSOR_MODE)KernelMode,    // IN
44371|         | KPROCESSOR_MODE AccessMode,
44372|         (PVOID *)&VirtualFileObject,    // OUT
44373|         | PVOID *Object,
44374|         NULL          // OUT
44375|         | POBJECT_HANDLE_INFORMATION HandleInformation //
44376|         | optional
44377|     );
44378|
44379| if(NT_SUCCESS(Status)) {
44380|     ZwClose(FileHandle);
44381|
44382|     SbTouchVolume(LiveVolName);
44383|
44384|     wcscpy(Buffer,LiveVolName);
44385|     RtlInitUnicodeString(&UniName,Buffer);
44386| }

```

```

44376|         InitializeObjectAttributes (
    | &ObjectAttributes,
44377|             &UniName,
44378|             | OBJ_CASE_INSENSITIVE,
44379|             NULL,
44380|             NULL );
44381|
44382|         Status = ZwCreateFile( &FileHandle,
44383|             GENERIC_READ,
    | // desired access
44384|             &ObjectAttributes,
    | // object attributes
44385|             &IoStatus,
44386|             NULL,
    | // alloc size
44387|             | FILE_ATTRIBUTE_NORMAL,    // file attributes
44388|             FILE_SHARE_WRITE |
    | FILE_SHARE_READ,                // share
    | access
44389|             FILE_OPEN,
    | // create disposition
44390|             0,                    //
    | create options
44391|             NULL, // eabuffer
44392|             0 ); // ealength
44393|         if(NT_SUCCESS(Status)) {
44394|             Status = ObReferenceObjectByHandle(
44395|                 FileHandle,    // IN HANDLE
    | Handle,
44396|                 GENERIC_READ,
44397|                 NULL,    // IN
    | POBJECT_TYPE ObjectType,    // optional
44398|                 (KPROCESSOR_MODE)KernelMode,
    | // IN KPROCESSOR_MODE AccessMode,
44399|                 (PVOID *)&LiveFileObject,    //
    | OUT PVOID *Object,
44400|                 NULL    // OUT
    | POBJECT_HANDLE_INFORMATION HandleInformation    //
    | optional
44401|             );
44402|
44403|             if(NT_SUCCESS(Status)) {
44404|
44405|                 LARGE_INTEGER LiveBitmapSize;
44406|                 LiveBitmapSize.QuadPart =
    | GetVolumeBitmapSize(LiveFileObject);
44407|
44408|                 if(LiveBitmapSize.QuadPart) {

```

```

44409|          FILE_FS_SIZE_INFORMATION
      | LiveFsSize;
44410|          ULONG Returned;
44411|
44412|          // get volume cluster size
44413|          Status =
      | IoQueryVolumeInformation(
44414|          LiveFileObject, // IN
      | PFILE_OBJECT FileObject,
44415|          FileFsSizeInformation, //
      | IN FS_INFORMATION_CLASS FsInformationClass,
44416|          sizeof(LiveFsSize), // IN
      | ULONG Length,
44417|          &LiveFsSize, // OUT PVOID
      | FsInformation,
44418|          &Returned // OUT PULONG
      | ReturnedLength
44419|          );
44420|          if(NT_SUCCESS(Status)) {
44421|          PDEVICE_OBJECT
      | DevObj=GetObjectFromName((WCHAR*)LiveVolName);
44422|          PFILTERED_EXTENSION DevExt
      | = GetFilteredExtension(DevObj);
44423|          LARGE_INTEGER
      | VolSizeInClusters;
44424|
44425|          ClusterSize =
      | LiveFsSize.SectorsPerAllocationUnit *
      | LiveFsSize.BytesPerSector;
44426|          VolSizeInClusters.QuadPart
      | = ( DevExt->Pi.PartitionLength.QuadPart + ClusterSize-1
      | )/ClusterSize ;
44427|
44428|          ASSERT
      | (VolSizeInClusters.QuadPart >=
      | LiveFsSize.TotalAllocationUnits.QuadPart);
44429|
44430|          Debug(DEBUG_DEVSUP,("Volume
      | '%S' has %!64d          clusters and %d
      | cluster size\n"
44431|          ,LiveVolName
44432|          | ,VolSizeInClusters.QuadPart
44433|          | ,LiveFsSize.SectorsPerAllocationUnit
44434|          ));
44435|
      | Debug(DEBUG_DEVSUP,("FileSys'%S' has %!64d (%d bitmap
      | bytes) clusters and %d cluster size\n"
44436|          ,VirtualVolName

```

```

44437|
| ,LiveFsSize.TotalAllocationUnits.QuadPart
44438| ,LiveBitmapSize.LowPart
44439|
| ,LiveFsSize.SectorsPerAllocationUnit
44440| ));
44441|
44442| #if DO_ALL_BITMAPS
44443| if (*BitMap!=NULL) {
44444| for (ULONG i=0;
| i*4<(((*BitMap)->SizeOfBitMap+7)/8;i+=0x8 ) {
44445| if ( (i==0) ||
44446| (
| (*BitMap)->Buffer[i+0] != (*BitMap)->Buffer[i-8+0] )
| ||
44447| (
| (*BitMap)->Buffer[i+1] != (*BitMap)->Buffer[i-8+1] )
| ||
44448| (
| (*BitMap)->Buffer[i+2] != (*BitMap)->Buffer[i-8+2] )
| ||
44449| (
| (*BitMap)->Buffer[i+3] != (*BitMap)->Buffer[i-8+3] )
| ||
44450| (
| (*BitMap)->Buffer[i+4] != (*BitMap)->Buffer[i-8+4] )
| ||
44451| (
| (*BitMap)->Buffer[i+5] != (*BitMap)->Buffer[i-8+5] )
| ||
44452| (
| (*BitMap)->Buffer[i+6] != (*BitMap)->Buffer[i-8+6] )
| ||
44453| (
| (*BitMap)->Buffer[i+7] != (*BitMap)->Buffer[i-8+7] ) )
| {
44454|
44455| | Debug(DEBUG_DEVSUP,(" Granule %08x: %08x %08x %08x
| %08x - %08x %08x %08x %08x\n"
44456| ,i*0x20
44457|
| ,(*BitMap)->Buffer[i+0x0]
44458| | ,(i+1)*4<(((*BitMap)->SizeOfBitMap+7)/8 ?
| (*BitMap)->Buffer[i+0x1] : 0x0dd1b1d5
44459| | ,(i+2)*4<(((*BitMap)->SizeOfBitMap+7)/8 ?
| (*BitMap)->Buffer[i+0x2] : 0x0dd1b1d5
44460|

```

```

    | ,(i+3)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x3] : 0x0dd1b1d5
44461|
    | ,(i+4)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x4] : 0x0dd1b1d5
44462|
    | ,(i+5)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x5] : 0x0dd1b1d5
44463|
    | ,(i+6)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x6] : 0x0dd1b1d5
44464|
    | ,(i+7)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x7] : 0x0dd1b1d5
44465|                                     ));
44466|                                     }
44467|                                     }
44468|                                     }
44469|     #endif
44470|     Status =
    | ReassessUsedGranuleBitMap (
44471|         VirtualFileObject,
44472|         VolSizeInClusters,
44473|         ClusterSize,
44474|         (PRTL_BITMAP*)BitMap );
44475|     if(NT_SUCCESS(Status)) {
44476|
    | Debug(DEBUG_DEVSUP,("Success creating 0x%x granule
    | bitmap\n",(*BitMap)->SizeOfBitMap));
44477|     #if DO_ALL_BITMAPS
44478|         if (*BitMap!=NULL) {
44479|             for (ULONG i=0;
    | i*4<((*BitMap)->SizeOfBitMap+7)/8;i+=0x8 ) {
44480|                 if ( (i==0) ||
44481|                     (
    | (*BitMap)->Buffer[i+0] != (*BitMap)->Buffer[i-8+0] )
    | ||
44482|                     (
    | (*BitMap)->Buffer[i+1] != (*BitMap)->Buffer[i-8+1] )
    | ||
44483|                     (
    | (*BitMap)->Buffer[i+2] != (*BitMap)->Buffer[i-8+2] )
    | ||
44484|                     (
    | (*BitMap)->Buffer[i+3] != (*BitMap)->Buffer[i-8+3] )
    | ||
44485|                     (
    | (*BitMap)->Buffer[i+4] != (*BitMap)->Buffer[i-8+4] )
    | ||
44486|                     (

```

```

    | (*BitMap)->Buffer[i+5] != (*BitMap)->Buffer[i-8+5] )
    | ||
44487|          (
    | (*BitMap)->Buffer[i+6] != (*BitMap)->Buffer[i-8+6] )
    | ||
44488|          (
    | (*BitMap)->Buffer[i+7] != (*BitMap)->Buffer[i-8+7] ) )
    | {
44489|
44490|    | Debug(DEBUG_DEVSUP,(" Granule %08x: %08x %08x %08x
    | %08x - %08x %08x %08x %08x\n"
44491|          ,i*0x20
44492|
    | ,(*BitMap)->Buffer[i+0x0]
44493|
    | ,(i+1)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x1] : 0x0dd1b1d5
44494|
    | ,(i+2)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x2] : 0x0dd1b1d5
44495|
    | ,(i+3)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x3] : 0x0dd1b1d5
44496|
    | ,(i+4)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x4] : 0x0dd1b1d5
44497|
    | ,(i+5)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x5] : 0x0dd1b1d5
44498|
    | ,(i+6)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x6] : 0x0dd1b1d5
44499|
    | ,(i+7)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
    | (*BitMap)->Buffer[i+0x7] : 0x0dd1b1d5
44500|          ));
44501|          }
44502|          }
44503|          }
44504|    #endif
44505|          } else {
44506|
    | Debug(DEBUG_DEVSUP,("Error %08x getting granule
    | bitmap\n",Status));
44507|          }
44508|          } else {
44509|          Debug(DEBUG_DEVSUP,("Error
    | %08x getting volume cluster size\n",Status));
44510|          }

```



```

44511|         } else {
44512|             Debug(DEBUG_DEVSUP,("Error
| getting size of bitmap\n"));
44513|             Status = STATUS_INVALID_HANDLE;
44514|         }
44515|         | ObDereferenceObject(LiveFileObject);
44516|     } else {
44517|         Debug(DEBUG_DEVSUP,("Error %08x
| getting live fileobject\n",Status));
44518|     }
44519|     ZwClose(FileHandle);
44520| } else {
44521|     Debug(DEBUG_DEVSUP,("Error %08x getting
| live filehandle\n",Status));
44522| }
44523|     ObDereferenceObject(VirtualFileObject);
44524| } else {
44525|     Debug(DEBUG_DEVSUP,("Error %08x getting
| virtual fileobject\n",Status));
44526| }
44527| } else {
44528|     Debug(DEBUG_DEVSUP,("Error %08x getting virtual
| filehandle\n",Status));
44529| }
44530| MemFreeString(Buffer);
44531| return Status;
44532| #else
44533| return 0;
44534| #endif
44535| }
44536|
44537| NTSTATUS FS_SystemCall (
44538|     PFILE_OBJECT FileObject,
44539|     ULONG IoControlCode,
44540|     const char *CallerFunctionName )
44541| {
44542|     PIRP Irp=NULL;
44543|     KEVENT Event={0};
44544|     IO_STATUS_BLOCK IoStatusBlock={0};
44545|     NTSTATUS Status=0;
44546|     PIO_STACK_LOCATION IrpStack=NULL;
44547|     PDEVICE_OBJECT DeviceObject;
44548|
44549|     __try {
44550|
44551|         Debug(DEBUG_DEVSUP,("%s
| (IoControlCode=%08x)\n",CallerFunctionName,IoControlCode
| ));
44552|

```

```

44553|     DeviceObject =
    | IoGetRelatedDeviceObject(FileObject);
44554|
44555|     //
44556|     // Set the event object to the unsigned
    | state.
44557|     // It will be used to signal request
    | completion.
44558|     //
44559|
44560|     KeInitializeEvent(&Event, NotificationEvent,
    | FALSE);
44561|
44562|
44563|     //
44564|     // Create IRP for read
44565|     //
44566|
44567|     Irp = IrpAllocateIrp(DeviceObject->StackSize);
44568|
44569|     if (!Irp) {
44570|         Debug(DEBUG_DEVSUP, ("%s: Error! Unable to
    | allocate irp\n", CallerFunctionName));
44571|         try_return(Status =
    | STATUS_INSUFFICIENT_RESOURCES);
44572|     }
44573|
44574|     Irp->Flags = 0; //IRP_READ_OPERATION;
44575|
44576|     Irp->AssociatedIrp.SystemBuffer = NULL;
44577|     Irp->MdlAddress = NULL;
44578|     Irp->UserBuffer = NULL;
44579|     Irp->UserEvent = &Event;
44580|     Irp->UserIoSb = &IoStatusBlock;
44581|     /*lint -save -e740 */
44582|     Irp->Tail.Overlay.Thread =
    | PsGetCurrentThread();
44583|     /*lint -restore */
44584|     Irp->Tail.Overlay.OriginalFileObject = NULL;
44585|     Irp->RequestorMode =
    | (KPROCESSOR_MODE)KernelMode;
44586|     Irp->IoStatus.Status =
    | STATUS_PENDING;
44587|     Irp->IoStatus.Information = 0;
44588|
44589|     IrpStack = IoGetNextIrpStackLocation(Irp);
44590|     RtlZeroMemory((PVOID)IrpStack,
    | sizeof(IO_STACK_LOCATION));
44591|     // just sets event
44592|     IoSetCompletionRoutine(Irp,

```

```

    | Sblo_ReadDeviceMdiCompletionRoutine, NULL, TRUE, TRUE,
    | TRUE);
44593|     IrpStack->MajorFunction =
    | IRP_MJ_FILE_SYSTEM_CONTROL;
44594|     IrpStack->MinorFunction =
    | IRP_MN_USER_FS_REQUEST;
44595|
    | IrpStack->Parameters.DeviceIoControl.IoControlCode =
    | IoControlCode;
44596|
    | IrpStack->Parameters.DeviceIoControl.OutputBufferLength
    | = 0;
44597|
    | IrpStack->Parameters.DeviceIoControl.InputBufferLength
    | = 0;
44598|
    | IrpStack->Parameters.DeviceIoControl.Type3InputBuffer =
    | NULL;
44599|     IrpStack->DeviceObject =
    | FileObject->DeviceObject;
44600|     IrpStack->FileObject = FileObject;
44601|
44602|     Status = IoCallDriver(DeviceObject, Irp);
44603|
44604|     if (Status == STATUS_PENDING) {
44605|
44606|         Debug(DEBUG_DEVSUP, ("%s: Waiting for
    | request to finish\n", CallerFunctionName));
44607|         ASSERT(KernelGetCurrentIrql() <
    | DISPATCH_LEVEL);
44608|         pmWaitForSingleObject(&Event, NULL);
44609|
44610|         Status = IoStatusBlock.Status;
44611|     }
44612|     IrpFreeIrp(Irp);
44613|
44614| try_exit: NOTHING;
44615| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
44616|     Status = GetExceptionCode();
44617|     Debug(DEBUG_DEVSUP, ("%s: Exception
    | %08x\n", CallerFunctionName, Status));
44618| }
44619|
44620| return Status;
44621| }
44622|
44623|
44624| NTSTATUS FillInWriteRequest( tWriteRequest
    | *WriteRequest, PDEVICE_OBJECT DeviceObject, PIRP Irp,

```

```

    | ULONG BPS )
44625| {
44626|   PIO_STACK_LOCATION currentIrpStack =
    | IoGetCurrentIrpStackLocation(Irp);
44627|
44628|   WriteRequest->DeviceObject = DeviceObject;
44629|   WriteRequest->Irp          = Irp;
44630|   WriteRequest->ByteOffset   =
    | currentIrpStack->Parameters.Write.ByteOffset;
44631|   WriteRequest->ByteLength   =
    | currentIrpStack->Parameters.Write.Length;
44632|   WriteRequest->RealSector.QuadPart =
    | WriteRequest->ByteOffset.QuadPart / BPS;
44633|   WriteRequest->RealCount      =
    | WriteRequest->ByteLength / BPS;
44634|
44635|   #define SectorSize (BPS)
44636|   WriteRequest->RoundedSector.QuadPart =
44637|
    | ROUND_DOWN(WriteRequest->RealSector.QuadPart,SECTORS_PER
    | _GRANULE);
44638|   WriteRequest->RoundedCount =
44639|       (ULONG)
    | (ROUND_UP(WriteRequest->RealSector.QuadPart +
    | WriteRequest->RealCount,SECTORS_PER_GRANULE)
44640|       -
    | WriteRequest->RoundedSector.QuadPart);
44641|   WriteRequest->RoundedSectorInBytes.QuadPart =
    | WriteRequest->RoundedSector.QuadPart * BPS;
44642|   WriteRequest->RoundedCountInBytes.QuadPart =
    | (unsigned __int64)WriteRequest->RoundedCount * BPS;
44643|   WriteRequest->DeviceObject = DeviceObject;
44644|   WriteRequest->Buffer       = NULL;
44645|   #undef SectorSize
44646|   return STATUS_SUCCESS;
44647| }
44648|
44649| WCHAR NibbleToHexWChar( unsigned char In, BOOLEAN
    | TakeUpper )
44650| {
44651|   In = TakeUpper?(In >> 4):(In & 0xf);
44652|   return (WCHAR)( ( In > 9 )?(In - 10 + 'a'):(In +
    | '0') );
44653| }
44654|
44655| NTSTATUS BufferToHexWChar( PVOID Buffer, ULONG
    | NumBytes, PWCHAR Out, ULONG *OutSize)
44656| {
44657|   NTSTATUS Status = STATUS_SUCCESS;
44658|   ULONG i;

```

```

44659| unsigned char *In = (unsigned char*)Buffer;
44660|
44661| //Round to take whole number of D_words
44662| NumBytes = (NumBytes+3)/4*4;
44663|
44664| if (Out==NULL) {
44665|     *OutSize = (NumBytes*2+1)*sizeof(WCHAR);
44666| } else {
44667|     if (*OutSize < 2) {
44668|         // just give back bad status as not room to
         | put even an empty string
44669|         Status = STATUS_BUFFER_TOO_SMALL;
44670|
44671|     } else {
44672|
44673|         if ((NumBytes*2+1)*sizeof(WCHAR) > *OutSize
         | ) {
44674|             //not enough room for all we have pack
         | the limit and give bad status
44675|             NumBytes =
         | (*OutSize/sizeof(WCHAR)-1)/2;
44676|             Status = STATUS_BUFFER_OVERFLOW;
44677|         }
44678|         *OutSize = (NumBytes*2+1)*sizeof(WCHAR);
44679|
44680|         for (i=0;i<NumBytes;In++,i++,Out+=2) {
44681|             Out[0] = NibbleToHexWChar(In[0],1);
44682|             Out[1] = NibbleToHexWChar(In[0],0);
44683|         }
44684|         Out[0] = L'\0';
44685|     }
44686| }
44687| return Status;
44688| }
44689|
44690|
44691| NTSTATUS FS_GetLastClusterFromFileMap (
44692|     const RETRIEVAL_POINTERS_BUFFER *RP,
44693|     LARGE_INTEGER &LastVcn )
44694| {
44695|     NTSTATUS Status = STATUS_NOT_FOUND;
44696|     LastVcn.QuadPart = __int64(-1);
44697|
44698|     if ( !RP ) {
44699|         ASSERT(RP != NULL);
44700|         Status = STATUS_INVALID_PARAMETER;
44701|     } else {
44702|         if ( RP->ExtentCount > 0 ) {
44703|             LastVcn.QuadPart =
         | RP->Extents[RP->ExtentCount-1].NextVcn.QuadPart - 1;

```

```

44704|         Status = STATUS_SUCCESS;
44705|     }
44706| }
44707|
44708| return Status;
44709| }
44710|
44711|
44712| NTSTATUS FS_GetFileMap( PFILE_OBJECT FileObject,
    | STARTING_VCN_INPUT_BUFFER *SVIB,
    | RETRIEVAL_POINTERS_BUFFER *RP, ULONG RPSize )
44713| {
44714|     PIRP          Irp=NULL;
44715|     KEVENT         Event={0};
44716|     IO_STATUS_BLOCK IoStatusBlock={0};
44717|     NTSTATUS        Status=0;
44718|     PIO_STACK_LOCATION IrpStack=NULL;
44719|     PDEVICE_OBJECT   DeviceObject;
44720|     ULONG            Ret = 0;
44721|
44722|     __try {
44723|
44724|         DeviceObject =
            | IoGetRelatedDeviceObject(FileObject);
44725|
44726|         //
44727|         // Set the event object to the unsigaled
            | state.
44728|         // It will be used to signal request
            | completion.
44729|         //
44730|
44731|         KeInitializeEvent(&Event, NotificationEvent,
            | FALSE);
44732|
44733|
44734|         //
44735|         // Create IRP for read
44736|         //
44737|
44738|         Irp = IrpAllocateIrp(DeviceObject->StackSize);
44739|
44740|         if (!Irp) {
44741|             Debug(DEBUG_DEVSUP,("FS_GetFileMap: Error!
            | Unable to allocate irp\n"));
44742|             try_return(Status =
            | STATUS_INSUFFICIENT_RESOURCES);
44743|         }
44744|
44745|         Irp->Flags = 0; //IRP_READ_OPERATION;

```

```

44746|
44747|     Irp->AssociatedIrp.SystemBuffer    = NULL;
44748|     Irp->MdlAddress                    = NULL;
44749|     Irp->UserBuffer = RP;
44750|     Irp->UserEvent = &Event;
44751|     Irp->UserIoSb = &IoStatusBlock;
44752|     /*lint -save -e740 */
44753|     Irp->Tail.Overlay.Thread =
        | PsGetCurrentThread();
44754|     /*lint -restore */
44755|     Irp->Tail.Overlay.OriginalFileObject = NULL;
44756|     Irp->RequestorMode =
        | (KPROCESSOR_MODE)KernelMode;
44757|     Irp->IoStatus.Status              =
        | STATUS_PENDING;
44758|     Irp->IoStatus.Information          = 0;
44759|
44760|     IrpStack = IoGetNextIrpStackLocation(Irp);
44761|     RtlZeroMemory((PVOID)IrpStack,
        | sizeof(IO_STACK_LOCATION));
44762|     // just sets event
44763|     IoSetCompletionRoutine(Irp,
        | Sblo_ReadDeviceMdlCompletionRoutine, NULL, TRUE, TRUE,
        | TRUE);
44764|     IrpStack->MajorFunction =
        | IRP_MJ_FILE_SYSTEM_CONTROL;
44765|     IrpStack->MinorFunction =
        | IRP_MN_USER_FS_REQUEST;
44766|
        | IrpStack->Parameters.DeviceIoControl.IoControlCode =
        | FSCTL_GET_RETRIEVAL_POINTERS;
44767|
        | IrpStack->Parameters.DeviceIoControl.OutputBufferLength
        | = RPSize;
44768|
        | IrpStack->Parameters.DeviceIoControl.InputBufferLength
        | = sizeof(STARTING_VCN_INPUT_BUFFER);
44769|
        | IrpStack->Parameters.DeviceIoControl.Type3InputBuffer =
        | SVIB;
44770|     IrpStack->DeviceObject =
        | FileObject->DeviceObject;
44771|     IrpStack->FileObject = FileObject;
44772|
44773|     Status = IoCallDriver(DeviceObject, Irp);
44774|
44775|     if (Status == STATUS_PENDING) {
44776|
44777|         Debug(DEBUG_DEVSUP,("FS_GetFileMap: Waiting
        | for get bitmap to finish\n"));

```

```

44778|         ASSERT(KeGetCurrentIrql() <
| DISPATCH_LEVEL);
44779|         pmWaitForSingleObject(&Event,NULL);
44780|
44781|         Status = IoStatusBlock.Status;
44782|         Ret = Irp->IoStatus.Information;
44783|     }
44784|     IrpFreeIrp(Irp);
44785|
44786| #if DO_ALL_IO
44787|     Debug(DEBUG_DEVSUP,("FS_GetFileMap:
| Status=%08x, size=%08x (%x,%x), Wanted=%l64x,
| got=%l64x, count=%08x\n",
44788|         Status,
44789|         RPSize,
44790|         IoStatusBlock.Information,
44791|         Ret,
44792|         SVIB->StartingVcn,
44793|         RP->StartingVcn,
44794|         RP->ExtentCount
44795|     ));
44796| #endif
44797|
44798| try_exit: NOTHING;
44799| }
| __except(ExceptionFilter(GetExceptionInformation())) {
44800|     Status = GetExceptionCode();
44801|     Debug(DEBUG_DEVSUP,("FS_GetFileMap: Exception
| %08x\n",Status));
44802| }
44803|
44804| return Status;
44805| }
44806|
44807| PDEVICE_OBJECT
44808| GetPSMStorageFilterObject(
44809|     IN PVPB Vpb
44810| )
44811| {
44812|     // follow Vpb->RealDevice->AttachedDevice stack.
44813|     PDEVICE_OBJECT p;
44814|     ULONG Count=0;
44815|
44816|     __try {
44817|         p=Vpb->RealDevice;
44818|         while ( p ) {
44819|             // follow the devices we have allocated
44820|             PDEVICE_OBJECT DevObj =
| PSMANDriverObject->DeviceObject;
44821|

```



```

44822|         while ( DevObj ) {
44823|             if (( DevObj == p ) ||
| (Count++>0x10000)) {
44824|                 break;
44825|             }
44826|
44827|             DevObj = DevObj->NextDevice;
44828|         } // while(DevObj)
44829|
44830|         if ( DevObj ) {
44831|             break;
44832|         }
44833|         p=p->AttachedDevice;
44834|     }
44835| } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
44836|     Debug(DEBUG_SFILTER,("SFILTER: Exception %08x
| in GPSMSFO\n",GetExceptionCode()));
44837|     p = NULL;
44838| }
44839| return p;
44840| }
44841|
44842| PDEVICE_OBJECT GetOurObjectForFileObject( PFILE_OBJECT
| FileObject )
44843| {
44844|     return GetPSMStorageFilterObject(FileObject->Vpb);
44845| }
44846|
44847| ErrorCode
44848| OpenAFile(
44849|     WCHAR          *File,
44850|     HANDLE          &FileHandle,
44851|     PFILE_OBJECT    &FileObject,
44852|     HANDLE          &WaitHandle,
44853|     PVOID           &WaitObject )
44854| {
44855|     UNICODE_STRING  FullFileName={0};
44856|     OBJECT_ATTRIBUTES ObjectAttributes={0};
44857|     NTSTATUS         Status=STATUS_UNSUCCESSFUL;
44858|     IO_STATUS_BLOCK  IoStatus={0};
44859|     ULONG            ShareAccess = 0;
44860|
44861|     FileHandle = INVALID_HANDLE_VALUE;
44862|     WaitHandle = INVALID_HANDLE_VALUE;
44863|     FileObject = NULL;
44864|     WaitObject = NULL;
44865|
44866|     PAGED_CODE();
44867|

```

```

44868|   Debug(DEBUG_DICT,("pd::OpenAFile(%S)\n",File));
44869|
44870|   RtlInitUnicodeString( &FullFileName, File);
44871|
44872|   InitializeObjectAttributes ( &ObjectAttributes,
44873|                               &FullFileName,
44874|                               OBJ_CASE_INSENSITIVE,
44875|                               NULL,
44876|                               NULL );
44877|
44878|   Status = ZwCreateFile( &FileHandle,
44879|                         FILE_GENERIC_READ |
44880|                         | FILE_GENERIC_WRITE, // desired access
44881|                         &ObjectAttributes, // object
44882|                         | attributes
44883|                         &IoStatus,
44884|                         NULL,           //
44885|                         | alloc size
44886|                         FILE_ATTRIBUTE_HIDDEN,
44887|                         | // file attributes
44888|                         ShareAccess,
44889|                         | // share access
44890|                         FILE_OPEN, //
44891|                         | FILE_OVERWRITE_IF,           // create
44892|                         | disposition
44893|                         FILE_SYNCHRONOUS_IO_NONALERT
44894|                         | |
44895|                         FILE_NO_COMPRESSION |
44896|                         FILE_WRITE_THROUGH |
44897|                         | FILE_NO_INTERMEDIATE_BUFFERING,
44898|                         NULL, // eabuffer
44899|                         0 ); // ealength
44900|
44901|   if ( NT_SUCCESS(Status) ) {
44902|       // Get a Object handle so we can wait on
44903|       | requests...
44904|       Status = ObReferenceObjectByHandle(
44905|           FileHandle,
44906|           | // IN HANDLE Handle,
44907|           FILE_GENERIC_READ | FILE_GENERIC_WRITE,
44908|           NULL,
44909|           | // IN POBJECT_TYPE ObjectType, (optional)
44910|           (KPROCESSOR_MODE)KernelMode,
44911|           | // IN KPROCESSOR_MODE AccessMode,
44912|           (PVOID *)&FileObject,
44913|           | // OUT PVOID *Object,
44914|           NULL );
44915|       | // OUT POBJECT_HANDLE_INFORMATION HandleInformation

```

```

    | (optional)
44903|
44904|     if ( NT_SUCCESS(Status) ) {
44905|         Status =
            | SbGetAsyncEvent(WaitHandle,WaitObject);
44906|         if ( NT_SUCCESS(Status) ) {
44907|             ASSERT(IsValidHandle(FileHandle));
44908|             ASSERT(FileObject != NULL);
44909|             ASSERT(IsValidHandle(WaitHandle));
44910|             ASSERT(WaitObject != NULL);
44911|             return STATUS_SUCCESS;
44912|         } else {
44913|             Debug(DEBUG_DICT,("pd::OpenAFile:
            | SbGetAsyncEvent(WaitHandle,WaitObject) returned
            | %08x\n",Status));
44914|             if ( FileObject ) {
44915|                 ObDereferenceObject (FileObject);
44916|                 FileObject = NULL;
44917|             }
44918|         }
44919|     } else {
44920|         Debug(DEBUG_DICT,("pd::OpenAFile:
            | ObReferenceObjectByHandle() returned %08x\n",Status));
44921|     }
44922|
44923|     ZwClose(FileHandle);
44924|     FileHandle = INVALID_HANDLE_VALUE;
44925|     FileObject = NULL;
44926| } else {
44927|     Debug(DEBUG_DICT,("pd::OpenAFile:
            | ZwCreateFile() returned %08x\n",Status));
44928| }
44929|
44930| // Getting here means an error occurred.
44931|
44932| #ifdef DEBUG
44933|     Debug(DEBUG_DICT,("pd::OpenAFile: Error! Unable to
            | open file '%S' %08x
            | (%08x)\n",File,Status,IoStatus.Status));
44934|     ASSERT(!NT_SUCCESS(Status));
44935|     ASSERT(FileHandle == INVALID_HANDLE_VALUE);
44936|     ASSERT(FileObject == NULL);
44937|     ASSERT(WaitHandle == INVALID_HANDLE_VALUE);
44938|     ASSERT(WaitObject == NULL);
44939| #endif /*DEBUG*/
44940|
44941|     return Status;
44942| }
44943|
44944| //-----

```

```

| -----
44945|
44946| ErrorCode CloseAFile (
44947|     HANDLE      &FileHandle,
44948|     PFILE_OBJECT &FileObject,
44949|     HANDLE      &WaitHandle,
44950|     PVOID       &WaitObject )
44951| {
44952|     Debug(DEBUG_DICT,("pd::CloseAFile %08x, %08x, %08x,
| %08x\n",FileHandle,FileObject,WaitHandle,WaitObject));
44953|
44954|     ASSERT (IsValidHandle(FileHandle));
44955|     ASSERT (FileObject != NULL);
44956|
44957|     ZwClose(FileHandle);
44958|     FileHandle = INVALID_HANDLE_VALUE;
44959|     if ( FileObject ) {
44960|         ObDereferenceObject(FileObject);
44961|         FileObject = NULL;
44962|     }
44963|
44964|     ASSERT (IsValidHandle(WaitHandle));
44965|     ASSERT (WaitObject != NULL);
44966|
44967|     ZwClose (WaitHandle);
44968|     WaitHandle = INVALID_HANDLE_VALUE;
44969|     if ( WaitObject ) {
44970|         ObDereferenceObject (WaitObject);
44971|         WaitObject = NULL;
44972|     }
44973|
44974|     return 0;
44975| }
44976|
44977| /*
44978|     FileObject is any open file on the specified volume
44979| */
44980| NTSTATUS FS_GetVolumeInfo( PFILE_OBJECT FileObject,
| ULONG &ClusterSize, ULONG &SectorSize, LARGE_INTEGER
| &TotalClusters, LARGE_INTEGER &AvailClusters )
44981| {
44982|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
44983|     POBJECT_NAME_INFORMATION OBI=NULL;
44984|     PFILE_OBJECT NewFileObject=NULL;
44985|     PDEVICE_OBJECT DeviceObject=NULL;
44986|     ULONG Returned=0;
44987|     OBJECT_ATTRIBUTES ObjAttr={0};
44988|     IO_STATUS_BLOCK IoStatus={0};
44989|     HANDLE VolumeHandle = INVALID_HANDLE_VALUE;
44990|

```

```

44991|    // FileObject->Vpb->DeviceObject == unnamed
      | filesystem object
44992|    // FileObject->Vpb->RealDevice == Volume as owned
      | by lowest device (eg \harddisk0\partition1)
44993|
44994|    if((FileObject->Vpb) && (FileObject->Vpb->Flags &
      | VPB_MOUNTED)) {
44995|        // we need to get a FileObject to the "Volume"
44996|
44997|        // step 1. We need the device name
      | (\harddisk0\partition1)
44998|
44999|        Status =
      | ObQueryNameString(FileObject->Vpb->RealDevice,NULL, 0,
      | &Returned );
45000|
45001|        if((Status==STATUS_INFO_LENGTH_MISMATCH) &&
      | (Returned>0)) {
45002|            OBI = (POBJECT_NAME_INFORMATION)
      | MemAllocatePoolWithTag(PagedPool,Returned+sizeof(WCHAR),
      | FILENAMETAG);
45003|            if(OBI) {
45004|                Status =
      | ObQueryNameString(FileObject->Vpb->RealDevice,OBI,Return
      | ed,&Returned );
45005|
45006|                if(NT_SUCCESS(Status)) {
45007|                    wcscat(OBI->Name.Buffer,L"\\");
45008|                    OBI->Name.Length+=sizeof(WCHAR);
45009|
45010|                    InitializeObjectAttributes (
      | &ObjAttr,&OBI->Name, OBJ_CASE_INSENSITIVE, NULL, NULL
      | );
45011|
45012|                    Status = ZwOpenFile(
45013|                        &VolumeHandle,
45014|                        0,
45015|                        &ObjAttr,
45016|                        &IoStatus,
45017|                        0,
45018|                        0
45019|                    );
45020|
45021|                    if(NT_SUCCESS(Status)) {
45022|                        // get file object from handle
45023|
      | ASSERT(IsValidHandle(VolumeHandle));
45024|                        Status =
      | ObReferenceObjectByHandle(
45025|                            VolumeHandle,    // IN

```

```

    | HANDLE Handle,
45026|         GENERIC_READ,
45027|         NULL,          // IN
    | POBJECT_TYPE ObjectType,      // optional
45028|
    | (KPROCESSOR_MODE)KernelMode,    // IN
    | KPROCESSOR_MODE AccessMode,
45029|         (PVOID *)&NewFileObject,
    | // OUT PVOID *Object,
45030|         NULL          // OUT
    | POBJECT_HANDLE_INFORMATION HandleInformation //
    | optional
45031|         );
45032|         if(NT_SUCCESS(Status)) {
45033|
45034|             FILE_FS_SIZE_INFORMATION
    | FsSize;
45035|             ULONG Returned;
45036|
45037|             // get volume cluster size
45038|             Status =
    | IoQueryVolumeInformation(
45039|                 NewFileObject, // IN
    | PFILE_OBJECT FileObject,
45040|                 FileFsSizeInformation,
    | // IN FS_INFORMATION_CLASS FsInformationClass,
45041|                 sizeof(FsSize), // IN
    | ULONG Length,
45042|                 &FsSize, // OUT PVOID
    | FsInformation,
45043|                 &Returned // OUT PULONG
    | ReturnedLength
45044|         );
45045|         if(NT_SUCCESS(Status)) {
45046|             ClusterSize =
    | FsSize.SectorsPerAllocationUnit*FsSize.BytesPerSector;
45047|             SectorSize =
    | FsSize.BytesPerSector;
45048|             TotalClusters =
    | FsSize.TotalAllocationUnits;
45049|             AvailClusters =
    | FsSize.AvailableAllocationUnits;
45050|
    | Debug(DEBUG_DEVSUP,("FS_GetVolumeInfo '%S' %08x %08x -
    | %!64x/%!64x\n",OBI->Name.Buffer,ClusterSize,SectorSize,A
    | vailClusters,TotalClusters));
45051|         } else {
45052|
    | Debug(DEBUG_DEVSUP,("IoQueryVolume failed with status
    | %08x\n",Status));

```

```

45053|         }
45054|         | ObDereferenceObject(NewFileObject);
45055|         } else {
45056|         Debug(DEBUG_DEVSUP,("ObRef
         | failed %08x\n",Status));
45057|         }
45058|         ZwClose(VolumeHandle);
45059|         VolumeHandle =
         | INVALID_HANDLE_VALUE;
45060|     } else {
45061|     | Debug(DEBUG_DEVSUP,("IoGetDevice failed
         | %08x\n",Status));
45062|     }
45063|     } else {
45064|     Debug(DEBUG_DEVSUP,("Query name
         | failed error %08x\n",Status));
45065|     }
45066|     FREE_POINTER(OBI);
45067|     } else {
45068|     Debug(DEBUG_DEVSUP,("Out of memory for
         | %d byte\n",Returned));
45069|     Status = STATUS_INSUFFICIENT_RESOURCES;
45070|     }
45071|     } else {
45072|     Debug(DEBUG_DEVSUP,("Query name failed
         | getting size error %08x\n",Status));
45073|     }
45074|     } else {
45075|     Debug(DEBUG_DEVSUP,("Devsup: FS_GetVolumeInfo:
         | volume for file object %08x not
         | mounted\n",FileObject));
45076|     }
45077|     return Status;
45078| }
45079|
45080| /*
45081| FileObject is any open file on the specified volume
45082| */
45083| NTSTATUS FS_GetVolumeAttributes( PFILE_OBJECT
         | FileObject, PFILE_FS_ATTRIBUTE_INFORMATION Attrib,
         | ULONG BufferLength )
45084| {
45085|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
45086|     POBJECT_NAME_INFORMATION OBI=NULL;
45087|     PFILE_OBJECT NewFileObject=NULL;
45088|     PDEVICE_OBJECT DeviceObject=NULL;
45089|     ULONG Returned=0;
45090|     OBJECT_ATTRIBUTES ObjAttr={0};

```

```

45091|   IO_STATUS_BLOCK IoStatus={0};
45092|   HANDLE VolumeHandle = INVALID_HANDLE_VALUE;
45093|
45094|   // FileObject->Vpb->DeviceObject == unnamed
   | filesystem object
45095|   // FileObject->Vpb->RealDevice == Volume as owned
   | by lowest device (eg \harddisk0\partition1)
45096|
45097|   if((FileObject->Vpb) && (FileObject->Vpb->Flags &
   | VPB_MOUNTED)) {
45098|       // we need to get a FileObject to the "Volume"
45099|
45100|       // step 1. We need the device name
   | (\harddisk0\partition1)
45101|
45102|       Status =
   | ObQueryNameString(FileObject->Vpb->RealDevice,NULL, 0,
   | &Returned );
45103|
45104|       if((Status==STATUS_INFO_LENGTH_MISMATCH) &&
   | (Returned>0)) {
45105|           OBI = (POBJECT_NAME_INFORMATION)
   | MemAllocatePoolWithTag(PagedPool,Returned+sizeof(WCHAR),
   | FILENAMETAG);
45106|           if(OBI) {
45107|               Status =
   | ObQueryNameString(FileObject->Vpb->RealDevice,OBI,Return
   | ed,&Returned );
45108|
45109|               if(NT_SUCCESS(Status)) {
45110|                   wscat(OBI->Name.Buffer,L"\\");
45111|                   OBI->Name.Length+=sizeof(WCHAR);
45112|
45113|                   InitializeObjectAttributes (
   | &ObjAttr,&OBI->Name, OBJ_CASE_INSENSITIVE, NULL, NULL
   | );
45114|
45115|                   Status = ZwOpenFile(
45116|                       &VolumeHandle,
45117|                       0,
45118|                       &ObjAttr,
45119|                       &IoStatus,
45120|                       0,
45121|                       0
45122|                   );
45123|
45124|                   if(NT_SUCCESS(Status)) {
45125|                       // get file object from handle
45126|
   | ASSERT(IsValidHandle(VolumeHandle));

```



```

45127|             Status =
| ObReferenceObjectByHandle(
45128|             VolumeHandle,      // IN
| HANDLE Handle,
45129|             GENERIC_READ,
45130|             NULL,              // IN
| POBJECT_TYPE ObjectType,      // optional
45131|
| (KPROCESSOR_MODE)KernelMode,  // IN
| KPROCESSOR_MODE AccessMode,
45132|             (PVOID *)&NewFileObject,
| // OUT PVOID *Object,
45133|             NULL              // OUT
| POBJECT_HANDLE_INFORMATION HandleInformation //
| optional
45134|         );
45135|         if(NT_SUCCESS(Status)) {
45136|
45137|             ULONG Returned;
45138|
45139|             // get volume cluster size
45140|             Status =
| IoQueryVolumeInformation(
45141|             NewFileObject, // IN
| PFILE_OBJECT FileObject,
45142|
| FileFsAttributeInformation, // IN FS_INFORMATION_CLASS
| FsInformationClass,
45143|             BufferLength, // IN
| ULONG Length,
45144|             Attrb, // OUT PVOID
| FsInformation,
45145|             &Returned // OUT PULONG
| ReturnedLength
45146|         );
45147|         if(NT_SUCCESS(Status)) {
45148|             } else {
45149|
| Debug(DEBUG_DEVSUP,("IoQueryVolume failed with status
| %08x\n",Status));
45150|             }
45151|
| ObDereferenceObject(NewFileObject);
45152|         } else {
45153|             Debug(DEBUG_DEVSUP,("ObRef
| failed %08x\n",Status));
45154|         }
45155|         ZwClose(VolumeHandle);
45156|         VolumeHandle =
| INVALID_HANDLE_VALUE;

```

```

45157|         } else {
45158|         | Debug(DEBUG_DEVSUP,("IoGetDevice failed
45159|         | %08x\n",Status));
45160|         }
45161|         Debug(DEBUG_DEVSUP,("Query name
45162|         | failed error %08x\n",Status));
45163|         }
45164|         FREE_POINTER(OBI);
45165|         Debug(DEBUG_DEVSUP,("Out of memory for
45166|         | %d byte\n",Returned));
45167|         Status = STATUS_INSUFFICIENT_RESOURCES;
45168|         } else {
45169|         Debug(DEBUG_DEVSUP,("Query name failed
45170|         | getting size error %08x\n",Status));
45171|         }
45172|         Debug(DEBUG_DEVSUP,("Devsup: FS_GetVolumeInfo:
45173|         | volume for file object %08x not
45174|         | mounted\n",FileObject));
45175|         }
45176|         return Status;
45177|     }
45178| NTSTATUS
45179| FS_SetEventCallBack(
45180|     IN PDEVICE_OBJECT DeviceObject,
45181|     IN PIRP Irp,
45182|     IN PVOID Context
45183| )
45184| {
45185|     NOT_REFERENCED(DeviceObject);
45186|     NOT_REFERENCED(Context);
45187|
45188|     pmSetEvent(Irp->UserEvent);
45189|
45190|     // keep nt from touching our irp, which will be
45191|     | handled by person
45192|     // who wanted the event set
45193|     return STATUS_MORE_PROCESSING_REQUIRED;
45194| }
45195| NTSTATUS FS_MoveFile( PFILE_OBJECT FileObject,
45196|     | MOVE_FILE_DATA *Data )
45197| {
45198|     PIRP Irp=NULL;

```

```

45198| KEVENT          Event={0};
45199| IO_STATUS_BLOCK  IoStatusBlock={0};
45200| NTSTATUS         Status=0;
45201| PIO_STACK_LOCATION IrpStack=NULL;
45202| PDEVICE_OBJECT    DeviceObject;
45203| ULONG            Ret = 0;
45204|
45205| __try {
45206|
45207|     DeviceObject =
45208|         | IoGetRelatedDeviceObject(FileObject);
45209|         //
45210|         // Set the event object to the unsigned
45211|         | state.
45212|         // It will be used to signal request
45213|         | completion.
45214|         //
45215|         KeInitializeEvent(&Event, NotificationEvent,
45216|             | FALSE);
45217|         //
45218|         // Create IRP for read
45219|         //
45220|
45221|         Irp = IrpAllocateIrp(DeviceObject->StackSize);
45222|
45223|         if (!Irp) {
45224|             Debug(DEBUG_DEVSUP,("FS_MoveFile: Error!
45225|             | Unable to allocate irp\n"));
45226|             try_return(Status =
45227|                 | STATUS_INSUFFICIENT_RESOURCES);
45228|         }
45229|
45230|         Irp->Flags = 0; //IRP_READ_OPERATION;
45231|
45232|         Irp->AssociatedIrp.SystemBuffer = Data;
45233|         Irp->MdlAddress = NULL;
45234|         Irp->UserBuffer = NULL;
45235|         Irp->UserEvent = &Event;
45236|         Irp->UserIosb = &IoStatusBlock;
45237|         /*lint -save -e740 */
45238|         Irp->Tail.Overlay.Thread =
45239|             | PsGetCurrentThread();
45240|         /*lint -restore */
45241|         Irp->Tail.Overlay.OriginalFileObject = NULL;
45242|         Irp->RequestorMode =
45243|             | (KPROCESSOR_MODE)KernelMode;

```

```

45240|     Irp->IoStatus.Status          =
      | STATUS_PENDING;
45241|     Irp->IoStatus.Information      = 0;
45242|
45243|     IrpStack = IoGetNextIrpStackLocation(Irp);
45244|     RtlZeroMemory((PVOID)IrpStack,
      | sizeof(IO_STACK_LOCATION));
45245|     // just sets event
45246|     IoSetCompletionRoutine(Irp,
      | FS_SetEventCallBack, NULL, TRUE, TRUE, TRUE);
45247|     IrpStack->MajorFunction =
      | IRP_MJ_FILE_SYSTEM_CONTROL;
45248|     IrpStack->MinorFunction =
      | IRP_MN_USER_FS_REQUEST;
45249|
      | IrpStack->Parameters.DeviceIoControl.IoControlCode =
      | FSCTL_MOVE_FILE;
45250|
      | IrpStack->Parameters.DeviceIoControl.OutputBufferLength
      | = 0;
45251|
      | IrpStack->Parameters.DeviceIoControl.InputBufferLength
      | = sizeof(MOVE_FILE_DATA);
45252|
      | IrpStack->Parameters.DeviceIoControl.Type3InputBuffer =
      | NULL;
45253|     IrpStack->DeviceObject =
      | FileObject->DeviceObject;
45254|     IrpStack->FileObject = FileObject;
45255|
45256|     Status = IoCallDriver(DeviceObject, Irp);
45257|
45258|     if (Status == STATUS_PENDING) {
45259|
45260|         Debug(DEBUG_DEVSUP,("FS_MoveFile: Waiting
      | for get bitmap to finish\n"));
45261|         ASSERT(KeGetCurrentIrql() <
      | DISPATCH_LEVEL);
45262|         pmWaitForSingleObject(&Event,NULL);
45263|
45264|         Status = IoStatusBlock.Status;
45265|         Ret = Irp->IoStatus.Information;
45266|     }
45267|     IrpFreeIrp(Irp);
45268|
45269| #if DO_ALL_IO
45270|     Debug(DEBUG_DEVSUP,("FS_MoveFile:
      | Status=%08x\n",Status));
45271| #endif
45272|

```

```

45273| try_exit: NOTHING;
45274| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
45275|     Status = GetExceptionCode();
45276|     Debug(DEBUG_DEVSUP,("FS_MoveFile: Exception
    | %08x\n",Status));
45277| }
45278|
45279| return Status;
45280| }
45281|
45282| NTSTATUS CloseVolumeByFileObject( HANDLE &VolumeHandle,
    | PFILE_OBJECT &VolumeFileObject)
45283| {
45284|     if(VolumeFileObject) {
45285|         ObDereferenceObject(VolumeFileObject);
45286|         VolumeFileObject = NULL;
45287|     }
45288|
45289|     if(IsValidHandle(VolumeHandle)) {
45290|         ZwClose(VolumeHandle);
45291|         VolumeHandle = INVALID_HANDLE_VALUE;
45292|     }
45293|     return STATUS_SUCCESS;
45294| }
45295|
45296| NTSTATUS OpenVolumeByFileObject( PFILE_OBJECT
    | FileObject, HANDLE &VolumeHandle, PFILE_OBJECT
    | &VolumeFileObject, ULONG Root )
45297| {
45298|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
45299|     POBJECT_NAME_INFORMATION OBI=NULL;
45300|     ULONG Returned=0;
45301|     OBJECT_ATTRIBUTES ObjAttr={0};
45302|     IO_STATUS_BLOCK IoStatus={0};
45303|
45304|     VolumeHandle = INVALID_HANDLE_VALUE;
45305|     VolumeFileObject = NULL;
45306|
45307|     // FileObject->Vpb->DeviceObject == unnamed
    | filesystem object
45308|     // FileObject->Vpb->RealDevice == Volume as owned
    | by lowest device (eg \harddisk0\partition1)
45309|
45310|     if((FileObject->Vpb) && (FileObject->Vpb->Flags &
    | VPB_MOUNTED)) {
45311|         // we need to get a FileObject to the "Volume"
45312|
45313|         // step 1. We need the device name
    | (\harddisk0\partition1)

```



```

45347|                (PVOID *)&VolumeFileObject,
| // OUT PVOID *Object,
45348|                NULL // OUT
| POBJECT_HANDLE_INFORMATION HandleInformation //
| optional
45349|                );
45350|                if(NT_SUCCESS(Status)) {
45351|                    Debug(DEBUG_DEVSUP,("FO:
| %08x = Volume: %08x %08x -
| '%S'\n",FileObject,VolumeHandle,VolumeFileObject,OBI->Na
| me.Buffer));
45352|                    FREE_POINTER(OBI);
45353|                    ASSERT(VolumeFileObject !=
| NULL);
45354|                    return Status;
45355|                } else {
45356|                    Debug(DEBUG_DEVSUP,("ObRef
| failed %08x\n",Status));
45357|                }
45358|                    ZwClose(VolumeHandle);
45359|                    VolumeHandle =
| INVALID_HANDLE_VALUE;
45360|                    VolumeFileObject = NULL;
45361|                } else {
45362|                    Debug(DEBUG_DEVSUP,("IoGetDevice failed
| %08x\n",Status));
45363|                }
45364|            } else {
45365|                Debug(DEBUG_DEVSUP,("Query name
| failed error %08x\n",Status));
45366|            }
45367|                FREE_POINTER(OBI);
45368|            } else {
45369|                Debug(DEBUG_DEVSUP,("Out of memory for
| %d byte\n",Returned));
45370|                Status = STATUS_INSUFFICIENT_RESOURCES;
45371|            }
45372|        } else {
45373|            Debug(DEBUG_DEVSUP,("Query name failed
| getting size error %08x\n",Status));
45374|        }
45375|    } else {
45376|        Debug(DEBUG_DEVSUP,("Devsup:
| OpenVolumeByFileObject: volume for file object %08x not
| mounted\n",FileObject));
45377|    }
45378|
45379|    // Getting here means an error occurred
45380|    ASSERT(!NT_SUCCESS(Status));

```

```

45381|  ASSERT(VolumeHandle == INVALID_HANDLE_VALUE);
45382|  ASSERT(VolumeFileObject == NULL);
45383|  return Status;
45384| }
45385|
45386| #ifdef DEBUG
45387| NTSTATUS ExtendAFile()
45388| {
45389|  NTSTATUS Status=STATUS_UNSUCCESSFUL;
45390|  LARGE_INTEGER Initial;
45391|  PFILE_OBJECT FileObject;
45392|  HANDLE FileHandle,WaitHandle;
45393|  PVOID WaitObject;
45394|
45395| #define FileName L"\\??\\d:\\temp\\test"
45396|
45397|  Initial.QuadPart = 1024*1024*50;
45398|
45399|  SbDeleteFile(FileName, FILE_ATTRIBUTE_NORMAL);
45400|  SbCreateAndFillFile ( FileName, &Initial, NULL, 00,
    | FILLONWRITE_DISABLED);
45401|
45402|  Status =
    | OpenAFile(FileName,FileHandle,FileObject,WaitHandle,Wait
    | Object);
45403|
45404|  if(NT_SUCCESS(Status)) {
45405|    PFILE_OBJECT VolumeObject = NULL;
45406|    HANDLE VolumeHandle = INVALID_HANDLE_VALUE;
45407|
45408|    // open root object
45409|    Status =
    | OpenVolumeByFileObject(FileObject,VolumeHandle,VolumeObj
    | ect,TRUE);
45410|    if(NT_SUCCESS(Status)) {
45411|      FILE_FS_SIZE_INFORMATION FsSize;
45412|      ULONG Returned;
45413|      // get volume cluster size
45414|      Status = IoQueryVolumeInformation(
45415|        VolumeObject, // IN PFILE_OBJECT
    | FileObject,
45416|        FileFsSizeInformation, // IN
    | FS_INFORMATION_CLASS FsInformationClass,
45417|        sizeof(FsSize), // IN ULONG Length,
45418|        &FsSize, // OUT PVOID FsInformation,
45419|        &Returned // OUT PULONG ReturnedLength
45420|      );
45421|
45422|      if(NT_SUCCESS(Status)) {
45423|

```



```

    | CloseVolumeByFileObject(VolumeHandle,VolumeObject);
45424|
45425|         // open volume object
45426|         Status =
    | OpenVolumeByFileObject(FileObject,VolumeHandle,VolumeObj
    | ect,FALSE);
45427|         if ( NT_SUCCESS(Status) ) {
45428|             __int64 Size =
    | GetVolumeBitmapSize(VolumeObject);
45429|
45430|             if(Size) {
45431|                 STARTING_LCN_INPUT_BUFFER SLIB;
45432|                 ULONG
    | ByteSize=FIELD_OFFSET(VOLUME_BITMAP_BUFFER,Buffer)+(ULON
    | G)(Size & 0xffffffff);
45433|                 VOLUME_BITMAP_BUFFER *VB=
    | (VOLUME_BITMAP_BUFFER
    | *)MemAllocatePoolWithTag(PagedPool,ByteSize,TEMPTAG);
45434|                 if(VB) {
45435|                     ULONG
    | CountToFind=Initial.LowPart /
    | (FsSize.SectorsPerAllocationUnit*FsSize.BytesPerSector);
45436|                     SLIB.StartingLcn.QuadPart=
    | 0;
45437|                     Status =
    | FS_GetVolumeBitmap(VolumeObject,&SLIB,VB,ByteSize);
45438|                     if(NT_SUCCESS(Status)) {
45439|                         RTL_BITMAP BitMap;
45440|
    | ASSERT(VB->BitmapSize.HighPart == 0);
45441|
    | RtlInitializeBitMap(&BitMap,(ULONG*)VB->Buffer,VB->Bitma
    | pSize.LowPart);
45442|                     ULONG Pos =
    | RtlFindClearBitsAndSet(&BitMap,CountToFind,0);
45443|                     if(Pos!=-1) {
45444|                         MOVE_FILE_DATA
    | Move;
45445|                         // VCN = Virtual
    | Cluster number = Offset from start of file
45446|                         // LCN = Logical
    | cluster number = Offset from start of volume
45447|                         Move.FileHandle =
    | FileHandle;
45448|                         // set to last
    | cluster in file
45449|
    | Move.StartingVcn.QuadPart = 0;//
    | *FsSize.SectorsPerAllocationUnit*FsSize.BytesPerSector;
45450|

```

```

| Move.StartingLcn.QuadPart = (unsigned __int64)Pos;//
| *FsSize.SectorsPerAllocationUnit*FsSize.BytesPerSector;
45451|             Move.ClusterCount =
| CountToFind;
45452|             Status =
| FS_MoveFile(VolumeObject,&Move);
45453|
| if(NT_SUCCESS(Status)) {
45454|
| Debug(DEBUG_DEVCON,("Success moving file\n"));
45455|             } else {
45456|
| Debug(DEBUG_DEVCON,("Error %08x moving
| file\n",Status));
45457|             }
45458|             } else {
45459|             Status =
| STATUS_INVALID_PARAMETER;
45460|
| Debug(DEBUG_DEVCON,("Error %08x getting 1 MB
| buffer\n",Status));
45461|             }
45462|             } else {
45463|
| Debug(DEBUG_DEVCON,("ExtendAFile: Error %08x getting
| bitmap\n",Status));
45464|             }
45465|             MemFreePool(VB);
45466|             } else {
45467|             Status =
| STATUS_INSUFFICIENT_RESOURCES;
45468|             Debug(DEBUG_DEVCON,("Error
| allocating memory for volume bitmap\n"));
45469|             }
45470|             } else {
45471|             Status =
| STATUS_INVALID_PARAMETER;
45472|             Debug(DEBUG_DEVCON,("Error %08x
| getting size of volume bitmap\n",Status));
45473|             }
45474|             } else {
45475|             Debug(DEBUG_DEVCON,("ExtentAFile:
| Error %08x in
| OpenVolumeByFileObject(...,FALSE)\n",Status));
45476|             }
45477|             } else {
45478|             Debug(DEBUG_DEVCON,("Error %08x getting
| volume info\n",Status));
45479|             }
45480|

```

```

    | CloseVolumeByFileObject(VolumeHandle,VolumeObject);
45481|     } else {
45482|         Debug(DEBUG_DEVCON,("ExtendAFile: Error
    | %08x opening volume\n",Status));
45483|     }
45484|
    | CloseAFile(FileHandle,FileObject,WaitHandle,WaitObject);
45485|     } else {
45486|         Debug(DEBUG_DEVCON,("ExtendAFile: Error %08x
    | opening file\n",Status));
45487|     }
45488|     return Status;
45489| }
45490|
45491|
45492| NTSTATUS ExtendAFile2()
45493| {
45494|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
45495|     LARGE_INTEGER Initial;
45496|     PFILE_OBJECT FileObject;
45497|     HANDLE FileHandle,WaitHandle;
45498|     PVOID WaitObject;
45499|
45500| #define FileName L"\\??\\d:\\temp\\test"
45501|
45502|     Initial.QuadPart = 4096;
45503|
45504|     SbDeleteFile(FileName, FILE_ATTRIBUTE_NORMAL);
45505|     SbCreateAndFillFile ( FileName, &Initial, NULL, 00,
    | PSMANFillOnWrite);
45506|     Status =
    | OpenAFile(FileName,FileHandle,FileObject,WaitHandle,Wait
    | Object);
45507|
45508|     if(NT_SUCCESS(Status)) {
45509|         PFILE_OBJECT VolumeObject = NULL;
45510|         HANDLE VolumeHandle = INVALID_HANDLE_VALUE;
45511|         RETRIEVAL_POINTERS_BUFFER *RP = NULL;
45512|         STARTING_VCN_INPUT_BUFFER SVIB = {0};
45513|         ULONG CalcSize=0;
45514|
45515|         // get Last vcn
45516|         ULONG Count = 16;
45517|
45518|         do {
45519|             if(RP) {
45520|                 MemFreePool(RP);
45521|                 RP = NULL;
45522|             }
45523|             CalcSize =

```

```

    | FIELD_OFFSET(RETRIEVAL_POINTERS_BUFFER,Extents)+Count*si
    | zeof(RP->Extents);
45524|         RP =
    | (RETRIEVAL_POINTERS_BUFFER*)MemAllocatePoolWithTag(Paged
    | Pool,CalcSize,TEMPTAG);
45525|         if(RP) {
45526|             RtlZeroMemory(RP,CalcSize);
45527|             SVIB.StartingVcn.QuadPart=0;
45528|             Status = FS_GetFileMap(
45529|                 FileObject,
45530|                 &SVIB,
45531|                 RP,
45532|                 CalcSize);
45533|             Count*=2;
45534|         } else {
45535|             Status = STATUS_INSUFFICIENT_RESOURCES;
45536|         }
45537|     } while(Status == STATUS_BUFFER_OVERFLOW);
45538|
45539|     if(NT_SUCCESS(Status)) {
45540|         LARGE_INTEGER LastVCN;
45541|         Status =
    | FS_GetLastClusterFromFileMap(RP,LastVCN);
45542|         if(NT_SUCCESS(Status)) {
45543|             Debug(DEBUG_DEVSUP,("Last cluster =
    | %08l64x\n",LastVCN));
45544|         } else {
45545|             Debug(DEBUG_DEVSUP,("Error %08x getting
    | last cluster\n",Status));
45546|         }
45547|
45548|         // open root object
45549|         Status =
    | OpenVolumeByFileObject(FileObject,VolumeHandle,VolumeObj
    | ect,TRUE);
45550|         if(NT_SUCCESS(Status)) {
45551|             FILE_FS_SIZE_INFORMATION FsSize;
45552|             ULONG Returned;
45553|
45554|             // get volume cluster size
45555|             Status = IoQueryVolumeInformation(
45556|                 VolumeObject, // IN PFILE_OBJECT
    | FileObject,
45557|                 FileFsSizeInformation, // IN
    | FS_INFORMATION_CLASS FsInformationClass,
45558|                 sizeof(FsSize), // IN ULONG Length,
45559|                 &FsSize, // OUT PVOID
    | FsInformation,
45560|                 &Returned // OUT PULONG
    | ReturnedLength

```

```

45561|         );
45562|
45563|         if(NT_SUCCESS(Status)) {
45564|
45565|             | CloseVolumeByFileObject(VolumeHandle,VolumeObject);
45566|
45567|             // open volume object
45568|             Status =
45569|             | OpenVolumeByFileObject(FileObject,VolumeHandle,VolumeObj
45570|             | ect,FALSE);
45571|             if ( NT_SUCCESS(Status) ) {
45572|                 __int64 Size =
45573|                 | GetVolumeBitmapSize(VolumeObject);
45574|
45575|                 if(Size) {
45576|                     STARTING_LCN_INPUT_BUFFER
45577|                     | SLIB;
45578|                     ULONG
45579|                     | ByteSize=FIELD_OFFSET(VOLUME_BITMAP_BUFFER,Buffer)+(ULON
45580|                     | G)(Size & 0xffffffff);
45581|                     VOLUME_BITMAP_BUFFER *VB=
45582|                     | (VOLUME_BITMAP_BUFFER
45583|                     | *)MemAllocatePoolWithTag(PagedPool,ByteSize,TEMPTAG);
45584|                     if(VB) {
45585|                         ULONG CountToFind=10;
45586|
45587|                         | SLIB.StartingLcn.QuadPart= 0;
45588|                         Status =
45589|                         | FS_GetVolumeBitmap(VolumeObject,&SLIB,VB,ByteSize);
45590|                         if(NT_SUCCESS(Status))
45591|                         | {
45592|                             RTL_BITMAP BitMap;
45593|
45594|                             | ASSERT(VB->BitmapSize.HighPart == 0);
45595|
45596|                             | RtlInitializeBitMap(&BitMap,(ULONG*)VB->Buffer,VB->Bitma
45597|                             | pSize.LowPart);
45598|
45599|                             ULONG Pos =
45600|                             | RtlFindClearBitsAndSet(&BitMap,CountToFind,0);
45601|                             if(Pos!=-1) {
45602|                                 MOVE_FILE_DATA
45603|                                 | Move;
45604|
45605|                                 // VCN =
45606|                                 | Virtual Cluster number = Offset from start of file
45607|                                 // LCN =
45608|                                 | Logical cluster number = Offset from start of volume
45609|                                 Move.FileHandle
45610|                                 | = FileHandle;
45611|
45612|                                 // set to last
45613|                                 | cluster in file

```

```

45590|
| Move.StartingVcn.QuadPart = LastVCN.QuadPart;//
| *FsSize.SectorsPerAllocationUnit*FsSize.BytesPerSector;
45591|
| Move.StartingLcn.QuadPart = (unsigned __int64)Pos;//
| *FsSize.SectorsPerAllocationUnit*FsSize.BytesPerSector;
45592|
| Move.ClusterCount = CountToFind;
45593|
| Status =
| FS_MoveFile(VolumeObject,&Move);
45594|
| if(NT_SUCCESS(Status)) {
45595|
| Debug(DEBUG_DEVCON,("Success moving file\n"));
45596|
| } else {
45597|
| Debug(DEBUG_DEVCON,("Error %08x moving
| file\n",Status));
45598|
| }
45599|
| } else {
45600|
| Status =
| STATUS_INVALID_PARAMETER;
45601|
| Debug(DEBUG_DEVCON,("Error %08x getting 1 MB
| buffer\n",Status));
45602|
| }
45603|
| } else {
45604|
| Debug(DEBUG_DEVCON,("ExtendAFile: Error %08x getting
| bitmap\n",Status));
45605|
| }
45606|
| MemFreePool(VB);
45607|
| } else {
45608|
| Status =
| STATUS_INSUFFICIENT_RESOURCES;
45609|
| Debug(DEBUG_DEVCON,("Error allocating memory for volume
| bitmap\n"));
45610|
| }
45611|
| } else {
45612|
| Status =
| STATUS_INVALID_PARAMETER;
45613|
| Debug(DEBUG_DEVCON,("Error
| %08x getting size of volume bitmap\n",Status));
45614|
| }
45615|
| } else {
45616|
| Debug(DEBUG_DEVCON,("Error %08x
| in OpenVolumeByFileObject(...,FALSE)\n",Status));
45617|
| }
45618|
| } else {

```

```

45619|         Debug(DEBUG_DEVCON,("Error %08x
| getting volume info\n",Status));
45620|     }
45621|
| CloseVolumeByFileObject(VolumeHandle,VolumeObject);
45622|     } else {
45623|         Debug(DEBUG_DEVCON,("ExtendAFile: Error
| %08x opening volume\n",Status));
45624|     }
45625|     MemFreePool(RP);
45626| } else {
45627|     Debug(DEBUG_DEVCON,("ExtendAFile: Error
| %08x getting extents\n",Status));
45628| }
45629|
| CloseAFile(FileHandle,FileObject,WaitHandle,WaitObject);
45630| } else {
45631|     Debug(DEBUG_DEVCON,("ExtendAFile: Error %08x
| opening file\n",Status));
45632| }
45633| return Status;
45634| }
45635| #endif
45636|
45637| NTSTATUS PsmWriteToFile(
45638|     pPsmFileInfo Info,
45639|     PIO_STATUS_BLOCK IoStatus,
45640|     PCVOID Buffer,
45641|     ULONG ByteCount,
45642|     PLARGE_INTEGER Location,
45643|     BOOLEAN DirectIo,
45644|     PERESOURCE DirectAccessResource
45645| #ifdef DEBUG
45646|     ,BOOLEAN CheckForDeadLock
45647| #endif
45648| )
45649| {
45650|     NTSTATUS Status = STATUS_UNSUCCESSFUL;
45651|
45652| #ifdef DEBUG
45653|     if(CheckForDeadLock &&
| IsSnapShotAcquiredForWrite()) {
45654|         // the reason this is bad is that we have the
| writer lock
45655|         // and any io that needs to be PSMed, needs to
| acquire the reader lock
45656|         // thus producing a deadlock, to fix it, dont
| call with writer lock
45657|         // which you can do by spinning off whatever
| you are trying to do to a

```

```

45658|    // worker thread.
45659|    Debug(DEBUG_DCPSM,("PsmWriteToFile: Snapshot
    | resource acquired for write!\n"));
45660|    DbgBreakPoint();
45661|    }
45662| #endif
45663|
45664|    if (( DirectIo ) || (PSMDirectIOOptions &
    | PSM_DIRECTIO_FLAG_ALWAYS_DO_DIRECT)) {
45665|        pmAcquireReaderLock ( DirectAccessResource,
    | TRUE );
45666|        ASSERT ( Info->Direct != NULL );
45667|        if ( Info->Direct != NULL ) {
45668|            ASSERT ( Info->Direct->readyForDirectIo()
    | );
45669|            if ( Info->Direct->readyForDirectIo() ) {
45670| #if DO_ALL_IO
45671|                Debug(DEBUG_DEVSUP,("PsmWriteToFile:
    | DirectIo %08x location %08l64x,
    | count=%08x\n",Buffer,*Location,ByteCount));
45672| #endif
45673|                Status = Info->Direct->write ( Buffer,
    | *Location, ByteCount );
45674|            } else {
45675|                Debug(DEBUG_DEVSUP,("PsmWriteToFile:
    | Attempt to perform direct IO when Info->Direct is not
    | ready\n"));
45676|            }
45677|        } else {
45678|            Debug(DEBUG_DEVSUP,("PsmWriteToFile:
    | Attempt to perform direct IO when
    | Info->Direct==NULL\n"));
45679|        }
45680|        pmReleaseReaderLock ( DirectAccessResource );
45681|    } else {
45682| #if DO_ALL_IO
45683|        Debug(DEBUG_DEVSUP,("PsmWriteToFile: File %08x
    | location %08l64x,
    | count=%08x\n",Buffer,*Location,ByteCount));
45684| #endif
45685|        Status =
    | SbWriteAndWait(Info,IoStatus,Buffer,ByteCount,Location);
45686|    }
45687|
45688|    return Status;
45689| }
45690|
45691| NTSTATUS PsmReadFromFile(
45692|     pPsmFileInfo Info,
45693|     PIO_STATUS_BLOCK IoStatus,

```



```

45694|          PVOID Buffer,
45695|          ULONG ByteCount,
45696|          PLARGE_INTEGER Location,
45697|          BOOLEAN DirectIo,
45698|          PERESOURCE DirectAccessResource
45699|      )
45700| {
45701|     NTSTATUS Status = STATUS_UNSUCCESSFUL;
45702|
45703|
45704|     if (( DirectIo ) || (PSMDirectIOOptions &
45705|         | PSM_DIRECTIO_FLAG_ALWAYS_DO_DIRECT)) {
45706|         pmAcquireReaderLock ( DirectAccessResource,
45707|             | TRUE );
45708|         ASSERT ( Info->Direct != NULL );
45709|         if ( Info->Direct != NULL ) {
45710|             ASSERT ( Info->Direct->readyForDirectIo()
45711|                 | );
45712|             if ( Info->Direct->readyForDirectIo() ) {
45713|                 #if DO_ALL_IO
45714|                 Debug(DEBUG_DEVSUP,("PsmReadFromFile:
45715|                     | DirectIo %08x location %08!64x,
45716|                     | count=%08x\n",Buffer,*Location,ByteCount));
45717|                 #endif
45718|                 Status = Info->Direct->read ( Buffer,
45719|                     | *Location, ByteCount );
45720|             } else {
45721|                 Debug(DEBUG_DEVSUP,("PsmReadFromFile:
45722|                     | Attempt to perform direct IO when Info->Direct is not
45723|                     | ready\n"));
45724|             }
45725|         } else {
45726|             Debug(DEBUG_DEVSUP,("PsmReadFromFile:
45727|                 | Attempt to perform direct IO when
45728|                 | Info->Direct==NULL\n"));
45729|         }
45730|         pmReleaseReaderLock ( DirectAccessResource );
45731|     } else {
45732|         #if DO_ALL_IO
45733|         Debug(DEBUG_DEVSUP,("PsmReadFromFile: File
45734|             | %08x location %08!64x,
45735|             | count=%08x\n",Buffer,*Location,ByteCount));
45736|         #endif
45737|         Status =
45738|             | SbReadAndWait(Info,IoStatus,Buffer,ByteCount,Location);
45739|     }
45740|
45741|     return Status;
45742| }
45743|

```

```

45731| NTSTATUS UpdateDriverSubSystemStatus( DWORD Id, const
      | WCHAR *SubSystemName )
45732| {
45733|     WCHAR Buffer[100];
45734|
45735|     //
      | \registry\system\machine\System\currentcontrolset\servic
      | es\psman5
45736|
      | RtlCopyMemory(Buffer,gRegistryPath.Buffer,gRegistryPath.
      | Length);
45737|     Buffer[gRegistryPath.Length / 2] = 0;
45738|     wcscat(Buffer,L"\\persistent");
45739|
45740|     Debug(DEBUG_DEVSUP,("UpdateDriverSubSystemStatus:
      | Id=%08x, SubSystemName='%S'\n",Id,SubSystemName));
45741|
45742|     return
      | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,SubSy
      | stemName,REG_DWORD,&Id,sizeof(DWORD));
45743| }
45744|
45745| NTSTATUS UpdateGlobalStatus( DWORD Id )
45746| {
45747|     return
      | UpdateDriverSubSystemStatus(Id,L"GlobalStatus");
45748| }
45749|
45750| // returns true if at least one snapshot is always keep
45751| ULONG AreThereAlwaysKeepSnapShots()
45752| {
45753|     ULONG FoundOne=0;
45754|     PDEVICE_OBJECT DevObj;
45755|     PFILTERED_EXTENSION DevExt=NULL;
45756|     pkSnapshotEntry p;
45757|
45758|     __try {
45759|         DevObj = PSMAN_DRIVER_OBJECT->DeviceObject;
45760|         // go through all volumes looking for snapshots
45761|         while(DevObj != NULL) {
45762|
      | if(PsmGetObjectTypeInfo(DevObj)==OBJECT_FILTEREDDISK) {
45763|             DevExt = GetFilteredExtension(DevObj);
45764|
      | GetSnapshotForRead();
45765|             __try {
45766|
      | p=GetTopSnapshot(&DevExt->Snapshots);
45767|             while(p) {
45768|
45769|

```

```

    | if(p->MasterSnapShot->Priority==255) {
45770|         FoundOne = TRUE;
45771|         DoneWithSnapShot(p);
45772|         break;
45773|     }
45774|
    | p=GetNextSnapShot(&DevExt->SnapShots,p);
45775|     }
45776|     } __finally {
45777|         ReleaseSnapShotForRead();
45778|     }
45779| }
45780| if(FoundOne) {
45781|     break;
45782| }
45783| DevObj=DevObj->NextDevice;
45784| }
45785| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
45786|
    | Debug(DEBUG_DCPSPM,("AreThereAlwaysKeepSnapShots:
    | Exception %08x\n",GetExceptionCode()));
45787| }
45788|
45789| return FoundOne;
45790| }
45791|
45792| WCHAR *GetPerVolumeRegistry( PDEVICE_OBJECT Volume )
45793| {
45794|     ULONG Length=(gRegistryPath.Length/2)+40+1;
45795|     PFILTERED_EXTENSION DevExt =
        | GetFilteredExtension(Volume);
45796|     WCHAR *Buffer = (WCHAR*)MemAllocateString(Length);
45797|
45798|     if(Buffer) {
45799|
        | RtlCopyMemory(Buffer,gRegistryPath.Buffer,gRegistryPath.
        | Length);
45800|         Buffer[gRegistryPath.Length / 2] = 0;
45801|         wcscat(Buffer,L "\\");
45802|         wcscat(Buffer,DevExt->VolumeGuid);
45803|     }
45804|     ASSERT(Buffer);
45805|     return Buffer;
45806| }
45807|
45808| void FreePerVolumeRegistry( WCHAR *Buffer )
45809| {
45810|     ASSERT(Buffer);
45811|     MemFreeString(Buffer);

```

```

45812| }
45813|
45814| //-----
| -----
45815| ULONG IsClusterServer()
45816| {
45817|     WCHAR Buffer[256];
45818|
| wcsncpy(Buffer,L"\\Registry\\Machine\\Cluster\\Persistent
| StorageManager\\");
45819|     if (
| RtlCheckRegistryKey(RTL_REGISTRY_ABSOLUTE,Buffer)==STATU
| S_SUCCESS ) {
45820|         return TRUE;
45821|     }
45822|     return FALSE;
45823| }
45824|
45825|
45826|
45827| File Listing: DEVSUP.h
45828|
45829| typedef const void *PCVOID;
45830|
45831| #define SBPSMAN_BUG_CODE          (0xe0000000)
45832|
45833| #define SB_BUG_BPSNOT512          (0x00000001)
45834| #define SB_BUG_ATDISPATCH        (0x00000002)
45835| #define SB_BUG_INIT_FAILED        (0x00000003)
45836| #define SB_CACHE_FULL             (0x00000004)
45837| #define SB_BUG_PREDICT_CANT_OPEN_HEADER (0x00000001)
45838| #define SB_BUG_REBOOT_FAILED      (0x00000001)
45839|
45840| #define SB_BUG_FILE_SBPSMAN       (0x00100000)
45841| #define SB_BUG_WRITE_FILE         (0x00200000)
45842| #define SB_BUG_FILE_INIT          (0x00400000)
45843| #define SB_BUG_FILE_PREDICT       (0x00800000)
45844| #define SB_BUG_REVERT             (0x01000000)
45845|
45846| #define PSManBugCheck(File,Code,A,B,C) {
| KeBugCheckEx(SBPSMAN_BUG_CODE | (Code), (File) |
| __LINE__, A, B, C ); }
45847|
45848| ULONG DeviceIsBeingBackedUp( PDEVICE_OBJECT
| DeviceObject );
45849|
45850| void FailRequest ( pkSnapshotEntry Snapshot, NTSTATUS
| Error );
45851| void SbTrace ( UCHAR Code, ULONG Arg1, ULONG Arg2,
| ULONG Arg3, ULONG Arg4, PCHAR Msg );

```

```

45852| void SbTrace2 ( char *Code, ULONG Arg1, ULONG Arg2,
      | ULONG Arg3, ULONG Arg4, PCHAR Msg, char *File, ULONG
      | Line );
45853|
45854|
45855| NTSTATUS SbOpenCacheFileInAsyncMode (
45856|         const WCHAR *CacheFile,
45857|         HANDLE
      | *FileHandle,
45858|         PFILE_OBJECT
      | *FileObject,
45859|         ULONG      Options
45860|     );
45861|
45862| NTSTATUS SbOpenCacheFileInSyncMode (
45863|         const WCHAR *CacheFile,
45864|         HANDLE
      | *FileHandle,
45865|         PFILE_OBJECT
      | *FileObject,
45866|         ULONG      Options
45867|     );
45868|
45869| NTSTATUS SbGetAsyncEvent (
45870|     HANDLE      &EventHandle,
45871|     PVOID      &EventObject );
45872|
45873|
45874| void SbGetRegistrySettings ( IN PUNICODE_STRING
      | RegistryPath );
45875|
45876| NTSTATUS Sblo_ReadDevice( PDEVICE_OBJECT DeviceObject,
45877|         PLARGE_INTEGER ByteOffset,
45878|         ULONG      ByteCount,
45879|         char      *Buff);
45880|
45881| NTSTATUS Sblo_WriteDevice(
45882|     PDEVICE_OBJECT DeviceObject,
45883|     PLARGE_INTEGER ByteOffset,
45884|     ULONG      ByteCount,
45885|     const char  *Buff );
45886|
45887|
45888| NTSTATUS Sblo_ReadDeviceMdl( PDEVICE_OBJECT
      | DeviceObject,
45889|         PLARGE_INTEGER ByteOffset,
45890|         ULONG      ByteCount,
45891|         PIRP
      | OriginalIrp,
45892|         PMDL      Mdl);

```

```

45893|
45894| NTSTATUS Sblo_GetGeometry( PDEVICE_OBJECT
    | DeviceObject,
45895|          PDISK_GEOMETRY Geometry );
45896|
45897|
45898| NTSTATUS Sblo_DeviceIoControl( PDEVICE_OBJECT
    | DeviceObject,
45899|          ULONG IoctlCode,
45900|          char *InBuffer,
45901|          ULONG InBufferSize,
45902|          char *OutBuffer,
45903|          ULONG OutBufferSize,
45904|          ULONG *BytesReturned
45905|          );
45906|
45907| NTSTATUS Sblo_GetDriveLayout( PDEVICE_OBJECT
    | DeviceObject,
45908|          PDRIVE_LAYOUT_INFORMATION DriveLayoutInfo,
45909|          ULONG
    | DriveLayoutInfoSize );
45910| #ifdef DEBUG
45911| void Debug_DumpSector( char *Buffer, ULONG Size );
45912| #else
45913| #define Debug_DumpSector(b,s)
45914| #endif
45915|
45916| NTSTATUS CheckMediaLoaded ( PDEVICE_OBJECT
    | DeviceObject, PIRP Irp );
45917|
45918| pOT_USER FindPSMUser ( PEPROCESS Process, PETHREAD
    | Thread );
45919| void DeletePSMUser( pOT_USER User );
45920| void AddPSMUser( pOT_USER User );
45921| pOT_USER FindPSMUserByFileObject ( PFILE_OBJECT
    | FileObject );
45922| pOT_USER FindPSMUserBySnapShot ( pkSnapShotMaster
    | SnapShot );
45923| NTSTATUS FlushVolume( const WCHAR *Name );
45924|
45925| NTSTATUS Sblo_Flush( PDEVICE_OBJECT DeviceObject );
45926| NTSTATUS Sblo_FlushTopLevelObject( PDEVICE_OBJECT
    | MyObject );
45927| NTSTATUS Sblo_InvalidateVolumes ( PDEVICE_OBJECT
    | FSObject, HANDLE FileHandle );
45928| NTSTATUS InvalidateVolumes( PDEVICE_OBJECT Volume );
45929| NTSTATUS FlushVolumeObject( PDEVICE_OBJECT Volume );
45930| PDEVICE_OBJECT GetFSObjectFromVolumeObject (
    | PDEVICE_OBJECT Volume );

```

```

45931| NTSTATUS Sblo_LockVolume( const WCHAR *VolumeName );
45932| NTSTATUS Sblo_UnlockVolume( const WCHAR *VolumeName );
45933| NTSTATUS Sblo_DismountVolume( const WCHAR *VolumeName
    | );
45934| NTSTATUS SbDeleteFile( const WCHAR *FileName, ULONG
    | FileAttributes);
45935| NTSTATUS SbDeleteZeroLengthFile( const WCHAR
    | *FileName);
45936|
45937| pkSnapshotEntry GetTopSnapshot( PLIST_ENTRY ListHead );
45938| BOOLEAN InList( PLIST_ENTRY ListHead, tkSnapshotEntry
    | *Snapshot);
45939|
45940| pkSnapshotEntry GetNextSnapshot( PLIST_ENTRY ListHead,
    | pkSnapshotEntry Snapshot );
45941| pkSnapshotMaster GetSnapshotMaster( PLIST_ENTRY
    | ListEntry );
45942| pkSnapshotEntry
    | FindSnapshotEntryForDevice(pkSnapshotMaster Snapshot,
    | PDEVICE_OBJECT DeviceObject);
45943| pkSnapshotEntry GetTopSnapshotForMaster( PLIST_ENTRY
    | ListHead );
45944| pkSnapshotEntry GetNextSnapshotForMaster( PLIST_ENTRY
    | ListHead, pkSnapshotEntry Snapshot );
45945| void DoneWithSnapshot( pkSnapshotEntry Snapshot );
45946| void UseSnapshot( pkSnapshotEntry Snapshot );
45947| NTSTATUS SblsADirectory ( const WCHAR *CacheFileName );
45948|
45949| VOID
45950| PSManSyncFilterWithTarget(
45951|     IN PDEVICE_OBJECT FilterDevice,
45952|     IN PDEVICE_OBJECT TargetDevice
45953| );
45954| NTSTATUS
45955| PSManForwardIrpSynchronous(
45956|     IN PDEVICE_OBJECT DeviceObject,
45957|     IN PIRP Irp
45958| );
45959| NTSTATUS
45960| PSManIrpCompletion(
45961|     IN PDEVICE_OBJECT DeviceObject,
45962|     IN PIRP Irp,
45963|     IN PVOID Context
45964| );
45965| VOID
45966| PSManAddCounters(
45967|     IN OUT PDISK_PERFORMANCE TotalCounters,
45968|     IN PDISK_PERFORMANCE NewCounters
45969| );
45970|

```

```

45971|
45972| typedef struct _FILE_FS_SIZE_INFORMATION {
45973|     LARGE_INTEGER TotalAllocationUnits;
45974|     LARGE_INTEGER AvailableAllocationUnits;
45975|     ULONG SectorsPerAllocationUnit;
45976|     ULONG BytesPerSector;
45977| } FILE_FS_SIZE_INFORMATION, *PFILE_FS_SIZE_INFORMATION;
45978|
45979| typedef struct _FILE_FS_ATTRIBUTE_INFORMATION {
45980|     ULONG FileSystemAttributes;
45981|     LONG MaximumComponentNameLength;
45982|     ULONG FileSystemNameLength;
45983|     WCHAR FileSystemName[1];
45984| } FILE_FS_ATTRIBUTE_INFORMATION,
    | *PFILE_FS_ATTRIBUTE_INFORMATION;
45985|
45986|
45987| extern "C" {
45988|     NTKERNELAPI
45989|     NTSTATUS
45990|     IoQueryVolumeInformation(
45991|         IN PFILE_OBJECT FileObject,
45992|         IN FS_INFORMATION_CLASS FsInformationClass,
45993|         IN ULONG Length,
45994|         OUT PVOID FsInformation,
45995|         OUT PULONG ReturnedLength
45996|     );
45997| }
45998|
45999|
46000| char *CopyFileNameOnly( char *Buffer, char *Path );
46001| NTSTATUS SbGetFileSize ( const WCHAR *CacheFileName,
    | PLARGE_INTEGER FileSize );
46002| int CreateJunction( const PWCHAR LinkDirectory, const
    | PWCHAR LinkTarget, LARGE_INTEGER Time );
46003| int DeleteJunction( const PWCHAR Junction );
46004| NTSTATUS CreateMountPoint( const PWCHAR LinkDirectory,
    | const PWCHAR LinkTarget, LARGE_INTEGER Time );
46005|
46006| NTSTATUS SbTouchVolume ( const WCHAR *VolumeName );
46007|
46008| //-----
    | -----
    | -----
46009|
46010| const ULONG FILLONWRITE_DISABLED          =
    | 0;          // don't zero fill diff file, and use
    | index sector EOF optimizations.
46011| const ULONG FILLONWRITE_ALL              =
    | 1;          // zero fill everything upon creation.

```



```

46012| const ULONG FILLONWRITE_EOF          =
    | 2;          // write one byte at EOF to initialize
    | entire file (not recommended).
46013|
46014| // It seems nt4sp4 ntfs upon receiving a
    | ZwSetInformationFile with FileEndOfFileInformation
    | doesn't actually
46015| // extend the file until it is written to. Seems to
    | only set the "filesize". We need to have the file
    | completely
46016| // allocated or when we write to our cache file we will
    | hang..
46017| const ULONG FILLONWRITES_DEF = FILLONWRITE_DISABLED;
    | // if no registry option specified, use most efficient
    | settings
46018|
46019| NTSTATUS SbCreateAndFillFile (
46020|     const WCHAR      *FileName,
46021|     PLARGE_INTEGER InitialSize,
46022|     PVOID             AbortEvent,
46023|     BYTE              Pattern,
46024|     ULONG             FillOnWriteOption );
46025|
46026| //-----
    | -----
    | -----
46027|
46028| ULONG IsBeingProcessedEx ( tWriteRequest *Request );
46029| ULONG IsBeingProcessed ( tWriteRequest *Request );
46030| NTSTATUS FindAndProcessVirtualVolumeBitMap( const WCHAR
    | *VirtualVolName, const WCHAR *LiveVolName, PRTL_BITMAP
    | *BitMap, ULONG &ClusterSize );
46031| NTSTATUS SbCreateDirectory ( const WCHAR
    | *DirectoryName, PSECURITY_DESCRIPTOR SD, ULONG
    | Attributes );
46032| NTSTATUS SbDeleteMountPoint( const WCHAR *FileName );
46033| NTSTATUS SbDeleteAllReparsePointsAndDir( const WCHAR
    | *SnapShotsPathName );
46034| NTSTATUS SbSnapShotCleanup( const WCHAR
    | *SnapShotsPathName );
46035| NTSTATUS FillInWriteRequest( tWriteRequest
    | *WriteRequest, PDEVICE_OBJECT DeviceObject, PIRP Irp,
    | ULONG BPS );
46036| NTSTATUS SetFlushRoutine( PVOID UserModePointer );
46037|
46038| ErrorCode
46039| OpenAFile(
46040|     WCHAR      *File,
46041|     HANDLE     &FileHandle,
46042|     PFILE_OBJECT &FileObject,

```

```

46043| HANDLE      &WaitHandle,
46044| PVOID        &WaitObject );
46045|
46046| ErrorCode
46047| CloseAFile (
46048| HANDLE      &FileHandle,
46049| PFILE_OBJECT &FileObject,
46050| HANDLE      &WaitHandle,
46051| PVOID        &WaitObject );
46052|
46053| #define IO_REPARSE_TAG_MOUNT_POINT
      | (0xA0000003)
46054| #define FSCTL_SET_REPARSE_POINT
      | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 41, METHOD_BUFFERED,
      | FILE_SPECIAL_ACCESS) // REPARSE_DATA_BUFFER,
46055| #define FSCTL_GET_REPARSE_POINT
      | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 42, METHOD_BUFFERED,
      | FILE_ANY_ACCESS) // REPARSE_DATA_BUFFER
46056| #define FSCTL_DELETE_REPARSE_POINT
      | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 43, METHOD_BUFFERED,
      | FILE_SPECIAL_ACCESS) // REPARSE_DATA_BUFFER,
46057|
46058| //
46059| // Undocumented FSCTL_SET_REPARSE_POINT structure
      | definition
46060| //
46061| #define REPARSE_MOUNTPOINT_HEADER_SIZE 8
46062| typedef struct {
46063|     DWORD    ReparseTag;
46064|     DWORD    ReparseDataLength;
46065|     WORD     Reserved;
46066|     WORD     ReparseTargetLength;
46067|     WORD     ReparseTargetMaximumLength;
46068|     WORD     Reserved1;
46069|     WCHAR    ReparseTarget[1];
46070| } REPARSE_MOUNTPOINT_DATA_BUFFER,
      | *PREPARSE_MOUNTPOINT_DATA_BUFFER;
46071|
46072| NTSTATUS ShutdownSystem( SHUTDOWN_ACTION Action );
46073| NTSTATUS BufferToHexWChar( PVOID Buffer, ULONG
      | NumBytes, PWCHAR Out, ULONG *OutSize);
46074|
46075| #define FSCTL_GET_NTFS_VOLUME_DATA
      | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 25, METHOD_BUFFERED,
      | FILE_ANY_ACCESS) // NTFS_VOLUME_DATA_BUFFER
46076| #define FSCTL_GET_NTFS_FILE_RECORD
      | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 26, METHOD_BUFFERED,
      | FILE_ANY_ACCESS) // NTFS_FILE_RECORD_INPUT_BUFFER,
      | NTFS_FILE_RECORD_OUTPUT_BUFFER
46077| #define FSCTL_GET_VOLUME_BITMAP

```

```

    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 27, METHOD_NEITHER,
    | FILE_ANY_ACCESS) // STARTING_LCN_INPUT_BUFFER,
    | VOLUME_BITMAP_BUFFER
46078| #define FSCTL_GET_RETRIEVAL_POINTERS
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 28, METHOD_NEITHER,
    | FILE_ANY_ACCESS) // STARTING_VCN_INPUT_BUFFER,
    | RETRIEVAL_POINTERS_BUFFER
46079| #define FSCTL_MOVE_FILE
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 29, METHOD_BUFFERED,
    | FILE_SPECIAL_ACCESS) // MOVE_FILE_DATA,
46080| #define FSCTL_IS_VOLUME_DIRTY
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 30, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
46081| #define FSCTL_GET_HFS_INFORMATION
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 31, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
46082| #define FSCTL_ALLOW_EXTENDED_DASD_IO
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 32, METHOD_NEITHER,
    | FILE_ANY_ACCESS)
46083|
46084| #if(_WIN32_WINNT >= 0x0400)
46085| //
46086| // Structure for FSCTL_GET_VOLUME_BITMAP
46087| //
46088|
46089| typedef struct {
46090|
46091|     LARGE_INTEGER StartingLcn;
46092|
46093| } STARTING_LCN_INPUT_BUFFER,
    | *PSTARTING_LCN_INPUT_BUFFER;
46094|
46095| typedef struct {
46096|
46097|     LARGE_INTEGER StartingLcn;
46098|     LARGE_INTEGER BitmapSize;
46099|     UCHAR Buffer[1];
46100|
46101| } VOLUME_BITMAP_BUFFER, *PVOLUME_BITMAP_BUFFER;
46102| #endif /* _WIN32_WINNT >= 0x0400 */
46103|
46104| #if(_WIN32_WINNT >= 0x0400)
46105| //
46106| // Structure for FSCTL_GET_RETRIEVAL_POINTERS
46107| //
46108|
46109| typedef struct {
46110|
46111|     LARGE_INTEGER StartingVcn;
46112|

```

```

46113| } STARTING_VCN_INPUT_BUFFER,
      | *PSTARTING_VCN_INPUT_BUFFER;
46114|
46115| typedef struct RETRIEVAL_POINTERS_BUFFER {
46116|
46117|     ULONG ExtentCount;
46118|     LARGE_INTEGER StartingVcn;
46119|     struct {
46120|         LARGE_INTEGER NextVcn;
46121|         LARGE_INTEGER Lcn;
46122|     } Extents[1];
46123|
46124| } RETRIEVAL_POINTERS_BUFFER,
      | *PRETRIEVAL_POINTERS_BUFFER;
46125| #endif /* _WIN32_WINNT >= 0x0400 */
46126|
46127| #if(_WIN32_WINNT >= 0x0400)
46128| //
46129| // Structures for FSCTL_GET_NTFS_FILE_RECORD
46130| //
46131|
46132| typedef struct {
46133|
46134|     LARGE_INTEGER FileReferenceNumber;
46135|
46136| } NTFS_FILE_RECORD_INPUT_BUFFER,
      | *PNTFS_FILE_RECORD_INPUT_BUFFER;
46137|
46138| typedef struct {
46139|
46140|     LARGE_INTEGER FileReferenceNumber;
46141|     ULONG FileRecordLength;
46142|     UCHAR FileRecordBuffer[1];
46143|
46144| } NTFS_FILE_RECORD_OUTPUT_BUFFER,
      | *PNTFS_FILE_RECORD_OUTPUT_BUFFER;
46145| #endif /* _WIN32_WINNT >= 0x0400 */
46146|
46147| #if(_WIN32_WINNT >= 0x0400)
46148| //
46149| // Structure for FSCTL_MOVE_FILE
46150| //
46151|
46152| typedef struct {
46153|
46154|     HANDLE FileHandle;
46155|     LARGE_INTEGER StartingVcn;
46156|     LARGE_INTEGER StartingLcn;
46157|     ULONG ClusterCount;
46158|

```

```

46159| } MOVE_FILE_DATA, *PMOVE_FILE_DATA;
46160| #endif /* _WIN32_WINNT >= 0x0400 */
46161|
46162| NTSTATUS FS_GetVolumeBitmap(
46163|     PFILE_OBJECT      FileObject,
46164|     STARTING_LCN_INPUT_BUFFER *SLIB,
46165|     VOLUME_BITMAP_BUFFER *VB,
46166|     ULONG              VBSizeInBytes );
46167|
46168| NTSTATUS FS_GetFileMap( PFILE_OBJECT FileObject,
46169|     | STARTING_VCN_INPUT_BUFFER *SVIB,
46170|     | RETRIEVAL_POINTERS_BUFFER *RP, ULONG RPSize );
46171|
46172| NTSTATUS FS_GetLastClusterFromFileMap (
46173|     const RETRIEVAL_POINTERS_BUFFER *RP,
46174|     LARGE_INTEGER &LastVcn );
46175|
46176| PDEVICE_OBJECT GetOurObjectForFileObject( PFILE_OBJECT
46177|     | FileObject );
46178|
46179| PDEVICE_OBJECT
46180| GetPSMStorageFilterObject(
46181|     IN PVPB Vpb
46182| );
46183|
46184| NTSTATUS FS_GetVolumeInfo( PFILE_OBJECT FileObject,
46185|     | ULONG &ClusterSize, ULONG &SectorSize, LARGE_INTEGER
46186|     | &TotalClusters, LARGE_INTEGER &AvailClusters );
46187|
46188| NTSTATUS PsmWriteToFile(
46189|     pPsmFileInfo Info,
46190|     PIO_STATUS_BLOCK IoStatus,
46191|     PCVOID Buffer,
46192|     ULONG ByteCount,
46193|     PLARGE_INTEGER Location,
46194|     BOOLEAN DirectIo,
46195|     PERESOURCE DirectAccessResource
46196| );
46197| #ifdef DEBUG
46198|     ,BOOLEAN CheckForDeadLock=TRUE
46199| #endif
46200|
46201| );
46202|
46203| NTSTATUS PsmReadFromFile(
46204|     pPsmFileInfo Info,
46205|     PIO_STATUS_BLOCK IoStatus,
46206|     PVOID Buffer,
46207|     ULONG ByteCount,
46208|     PLARGE_INTEGER Location,
46209|     BOOLEAN DirectIo,
46210|     PERESOURCE DirectAccessResource

```

```

46204|         );
46205|
46206| NTSTATUS UpdateGlobalStatus( DWORD Id );
46207| NTSTATUS UpdateDriverSubSystemStatus( DWORD Id, const
    | WCHAR *SubSystemName );
46208| PSECURITY_DESCRIPTOR SbGetAdminOnlySD( );
46209| NTSTATUS Sblo_IsVolumeMounted( const WCHAR *VolumeName
    | );
46210| NTSTATUS FS_IsVolumeMounted ( PFILE_OBJECT FileObject
    | );
46211| ULONG AreThereAlwaysKeepSnapshots();
46212|
46213| NTSTATUS Sblo_OpenVolumeHandle (
46214|     const WCHAR     *VolumeName,
46215|     HANDLE           &VolumeHandle,
46216|     PFILE_OBJECT     &VolumeFileObject,
46217|     ULONG             DesiredAccess );
46218|
46219| NTSTATUS Sblo_CloseVolumeHandle (
46220|     HANDLE           &VolumeHandle,
46221|     PFILE_OBJECT     &VolumeFileObject );
46222|
46223| NTSTATUS FS_LockVolume   ( PFILE_OBJECT FileObject );
46224| NTSTATUS FS_UnlockVolume ( PFILE_OBJECT FileObject );
46225| NTSTATUS FS_DismountVolume ( PFILE_OBJECT FileObject );
46226| NTSTATUS ExtendFreeSpaceBitmaps( PDEVICE_OBJECT Volume,
    | LARGE_INTEGER NewSize );
46227| NTSTATUS Sblo_GetCapabilities( PDEVICE_OBJECT
    | DeviceObject,
46228|                                PIO SCSI_CAPABILITIES
    | Caps);
46229| NTSTATUS FS_GetVolumeAttributes( PFILE_OBJECT
    | FileObject, PFILE_FS_ATTRIBUTE_INFORMATION Attrib,
    | ULONG BufferLength );
46230|
46231| #define DWORD_ALIGN(x)      (((x)+31)/32)*32
46232|
46233| WCHAR *GetPerVolumeRegistry( PDEVICE_OBJECT Volume );
46234| void FreePerVolumeRegistry( WCHAR *Buffer );
46235| ULONG IsClusterServer();
46236|
46237|
46238|
46239| File Listing: FILE.cpp
46240|
46241| #include "precomp.h"
46242|
46243| CHAR *NtFsMetaDataFileNames[MAX_NTFS_ENTRY_NUM] = {
46244|     "$Mft",
46245|     "$MftMirr",

```

```

46246| "$LogFile",
46247| "$Volume",
46248| "$AttrDef",
46249| "$Root",
46250| "$Bitmap",
46251| "$Boot",
46252| "$BadClus",
46253| "$Secure",
46254| "$UpCase",
46255| "$Extend",
46256| "$12",
46257| "$13",
46258| "$14",
46259| "$15"
46260| };
46261|
46262| WCHAR *NtFsMetaDataFileNamesW[MAX_NTFS_ENTRY_NUM] = {
46263|     L"$Mft",
46264|     L"$MftMirr",
46265|     L"$LogFile",
46266|     L"$Volume",
46267|     L"$AttrDef",
46268|     L"$Root",
46269|     L"$Bitmap",
46270|     L"$Boot",
46271|     L"$BadClus",
46272|     L"$Secure",
46273|     L"$UpCase",
46274|     L"$Extend",
46275|     L"$12",
46276|     L"$13",
46277|     L"$14",
46278|     L"$15"
46279| };
46280|
46281|
46282| // this function retrieves the FULL path and filename
46283| // (which can be greater than MAX_PATH). If any
    | portions of the filename
46284| // are paged out, and this routine is called at
    | >=DISPATCH_LEVEL then
46285| // the filename will not be present. If called
    | <DISPATCH, the page
46286| // will be mapped in.
46287| // The returned Unicode string WILL be null terminated.
46288|
46289| /*
46290|     Filenames can look like these:
46291|
46292|     filename=test.file parent=null

```

```

46293|     filename=\test1\test2\test.file parent=null
46294|     filename=test2\test.file parent=\test1
      | parent=null
46295|     filename=test.file parent=test2 parent=\test1
      | parent=null
46296|
46297| */
46298|
46299| /*-----
      | -----*/
46300| STATIC PUNICODE_STRING GetUniString( ULONG Size )
46301| {
46302|     PUNICODE_STRING String;
46303|
46304|     PAGED_CODE();
46305|     String =
      | (PUNICODE_STRING)MemAllocatePoolWithTag(NonPagedPool,siz
      | eof(UNICODE_STRING)+Size+sizeof(WCHAR),FILENAMETAG);
46306|     if(String) {
46307|         RtlZeroMemory(String,
      | sizeof(UNICODE_STRING)+Size+sizeof(WCHAR));
46308|
46309|         String->Length      = (unsigned short)Size;
46310|         String->MaximumLength = (unsigned
      | short)(Size+sizeof(WCHAR));
46311|         /*lint -save -e740 */
46312|         String->Buffer      = (WCHAR*)(String+1);
46313|         /*lint -restore */
46314|     }
46315|     return String;
46316| }
46317|
46318| /*lint -save -e149 */
46319| /*-----
      | -----*/
46320| PUNICODE_STRING File_GetFileName(
46321|     PFILE_OBJECT fileObject
46322| )
46323| {
46324|     PUNICODE_STRING fileName=NULL;
46325|
46326|     // Do this in an exception handling block, in case
      | of mangled names in the
46327|     // file object
46328|     __try {
46329|
46330|         if ((fileObject) &&
      | (fileObject->FileName.Buffer)) {
46331|             ASSERT (fileObject->Type==IO_TYPE_FILE);
46332|             ASSERT

```



```

    | (fileObject->Size==sizeof(FILE_OBJECT));
46333|     ASSERT
    | (fileObject->FileName.MaximumLength>=fileObject->FileNam
    | e.Length);
46334|
46335|     fileName = GetUniString(
    | fileObject->FileName.Length );
46336|     if(fileName) {
46337|         __try {
46338|             // We need to move memory, not
    | strcpy as the can be no
46339|             // terminator (0).
46340|             RtlMoveMemory( fileName->Buffer,
    | fileObject->FileName.Buffer,
    | fileObject->FileName.Length);
46341|         } __except (
    | ExceptionFilter(GetExceptionInformation()) ) {
46342|             ULONG i;
46343|             // some kind of fault just
    | occurred. Probably accessing
46344|             // memory that has been paged out.
    | but can be a bad pointer
46345|             // also.
46346|             Debug(DEBUG_FILE,("Error! Fault
    | %08x while getting name\n",GetExceptionCode()));
46347|             // fill in with ?, so we know its
    | been mapped out.
46348|             for
    | (i=0;i<(fileObject->FileName.Length /
    | sizeof(WCHAR));i++)
46349|                 fileName->Buffer[i] = L'?';
46350|         }
46351|
46352|     }
46353|     return fileName;
46354| } else {
46355|     fileName = GetUniString( 4*sizeof(WCHAR) );
46356|     if(fileName) {
46357|         wcsncpy( fileName->Buffer, L"DASD" );
46358|     }
46359|     return fileName;
46360| }
46361|
46362|
46363| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
46364|     // Some kind of fault (probably a Page Fault
    | from a bad pointer)
46365|     // do cleanup...
46366|

```

```

46367|     Debug(DEBUG_FILE,("Error! Fault %08x in
| File_GetFileName\n",GetExceptionCode()));
46368|
46369|     if(fileName) {
46370|         FREE_POINTER(fileName);
46371|     }
46372|
46373|     fileName = GetUniString( 3*sizeof(WCHAR) );
46374|     if(fileName) {
46375|         wcsncpy( fileName->Buffer, L"???" );
46376|     }
46377|     return fileName;
46378| }
46379| }
46380| /*lint -restore */
46381|
46382| /*lint -save -e149 */
46383| /*-----
| -----*/
46384| PUNICODE_STRING File_GetFullFileName(
46385|     PFILE_OBJECT fileObject
46386| )
46387| {
46388|     ULONG          pathLen=0;
46389|     PWCHAR         pathOffset=NULL;
46390|     PFILE_OBJECT   relatedFileObject=NULL;
46391|     PUNICODE_STRING fullFileName=NULL;
46392|
46393|     // Do this in an exception handling block, in case
| of mangled names in the
46394|     // file object
46395|     __try {
46396|
46397|
46398|         fullFileName = GetUniString (
| MAX_PATH*sizeof(WCHAR) );
46399|         if(!fullFileName) {
46400|             Debug(DEBUG_FILE,("Error! Out of memory for
| FileName get,
| need=%d\n",sizeof(UNICODE_STRING)+MAX_PATH*sizeof(WCHAR)
| +sizeof(WCHAR)));
46401|             return NULL;
46402|         }
46403|
46404|         // Is it DASD (Volume) I/O?
46405|         if( (!fileObject) ||
46406|             (!fileObject->FileName.Buffer)
46407|         // || (fileObject->Flags &
| FO_DIRECT_DEVICE_OPEN)
46408|         ) {

```

```

46409| #if 0
46410|         // rob, 1-18-2001 - causes hangs to occur
         | when ntfs does a "checkpoint" of the volume.
46411|         ULARGE_INTEGER Id;
46412|
         | if(File_QueryMFTEntryNumber(fileObject,Id)==STATUS_SUCCE
         | SS) {
46413|             if(Id.QuadPart<MAX_NTFS_ENTRY_NUM) {
46414|
         | wcscpy(fullFileName->Buffer,NtFsMetaDataFileNamesW[Id.Lo
         | wPart]);
46415|             } else {
46416|                 swprintf( fullFileName->Buffer,
         | L"ID = %I64x", Id.QuadPart );
46417|             }
46418|             fullFileName->Length =
         | NumBytes(fullFileName->Buffer);
46419|             return fullFileName;
46420|         }
46421| #endif
46422|         wcscpy( fullFileName->Buffer, L"DASD" );
46423|         fullFileName->Length =
         | NumBytes(fullFileName->Buffer);
46424|         return fullFileName;
46425|     }
46426|
46427|     // find out how much room we need for the
         | entire path name
46428|     pathLen = 0;
46429|     relatedFileObject = fileObject;
46430|
46431|     while(relatedFileObject) {
46432|         // validity checks...
46433|
46434|         if (!MmIsAddressValid( relatedFileObject ))
         | {
46435|             Debug(DEBUG_FILE,("Error! FileObject %p
         | is not a valid pointer, (%p is the
         | root)\n",relatedFileObject,fileObject));
46436|             wcscpy(fullFileName->Buffer,L"Error!
         | Invalid FileObject");
46437|             fullFileName->Length =
         | NumBytes(fullFileName->Buffer);
46438|             return fullFileName;
46439|         }
46440|
46441|         if (relatedFileObject->Type!=IO_TYPE_FILE)
         | {
46442|             Debug(DEBUG_FILE,("Error! FileObject %p
         | is not a file object, (%p is the

```

```

    | root)\n",relatedFileObject,fileObject));
46443|         wcsncpy(fullFileName->Buffer,L"Error!
    | Not a FileObject");
46444|         fullFileName->Length =
    | NumBytes(fullFileName->Buffer);
46445|         return fullFileName;
46446|     }
46447|
46448| //         ASSERT (MmlsAddressValid(
    | relatedFileObject ));
46449| //         ASSERT
    | (relatedFileObject->Type==IO_TYPE_FILE);
46450|         ASSERT
    | (relatedFileObject->Size==sizeof(FILE_OBJECT));
46451|         ASSERT
    | (relatedFileObject->FileName.MaximumLength>=relatedFileO
    | bject->FileName.Length);
46452|
46453|         pathLen +=
    | relatedFileObject->FileName.Length + sizeof(WCHAR);
46454|         relatedFileObject =
    | relatedFileObject->RelatedFileObject;
46455|     }
46456|
46457|     // decrement the count by one (the root object
    | already has a slash
46458|     pathLen -= sizeof(WCHAR);
46459|
46460|     // due to FileName->Length being a USHORT and
    | being the number of
46461|     // bytes NOT chars (1 char = 2 bytes in
    | unicode), the largest filename
46462|     // can only 32768 Chars. So if the pathLen
    | (which is a ULONG) is
46463|     // greater than 65536 then something is wrong..
46464|     if (pathLen>65535) {
46465|         Debug(DEBUG_FILE,("Error! Invalid path
    | length for filename (%d)\n",pathLen));
46466|         wcsncpy(fullFileName->Buffer,L"Error!
    | Invalid Path Length");
46467|         fullFileName->Length =
    | NumBytes(fullFileName->Buffer);
46468|         return fullFileName;
46469|     }
46470|
46471|     // if greater than MAX_PATH (which was
    | allocated above),
46472|     // free and allocate how much we need.
46473|     if (pathLen>fullFileName->MaximumLength) {
46474|         FREE_POINTER(fullFileName);

```

```

46475|         fullFileName = GetUniString( pathLen );
46476|         if(!fullFileName) {
46477|             Debug(DEBUG_FILE,("Error! Out of memory
| for FileName get
| Need=%d\n",sizeof(UNICODE_STRING)+pathLen+sizeof(WCHAR))
| );
46478|             return NULL;
46479|         }
46480|     }
46481|
46482|     // ok, we now have a buffer big enough for the
| entire path and filename
46483|     // lets go ahead and put the path together by
| working backwards
46484|
46485|     fullFileName->Length = (unsigned short)pathLen;
46486|
46487|     // start at end and work backwards.
46488|     pathOffset = fullFileName->Buffer+(pathLen /
| sizeof(WCHAR));
46489|
46490|     // add the terminator char
46491|     pathOffset[0] = L'\0';
46492|     relatedFileObject = fileObject;
46493|     while(relatedFileObject) {
46494|         pathOffset -=
| (relatedFileObject->FileName.Length / sizeof(WCHAR));
46495|
46496|         __try {
46497|             // We need to move memory, not strcpy
| as the can be no
46498|             // terminator (0).
46499|             RtlMoveMemory( pathOffset,
| relatedFileObject->FileName.Buffer,
| relatedFileObject->FileName.Length);
46500|         } __except (
| ExceptionFilter(GetExceptionInformation()) ) {
46501|             ULONG i;
46502|             // some kind of fault just occurred.
| Probably accessing
46503|             // memory that has been paged out. but
| can be a bad pointer
46504|             // also.
46505|             Debug(DEBUG_FILE,("Error! Fault %08x
| while getting name\n",GetExceptionCode()));
46506|             // fill in with ?, so we know its been
| mapped out.
46507|             for
| (i=0;i<(relatedFileObject->FileName.Length /
| sizeof(WCHAR));i++)

```

```

46508|         pathOffset[i] = L'?';
46509|     }
46510|     // if not the root... (root == pathOffset
    | == start)
46511|     // we need to add a slash in front of the
    | name (because it is
46512|     // relative to the next file object
46513|     if(pathOffset>fullFileName->Buffer) {
46514|         // add in the separator
46515|         pathOffset--;
46516|         pathOffset[0] = L'\\';
46517|     }
46518|
46519|     relatedFileObject =
    | relatedFileObject->RelatedFileObject;
46520|
46521|     }
46522|
46523| #ifdef DEBUG
46524|     if ( ( ( ((char*)(fullFileName->Buffer))[0] )
    | == '?' ) ||
46525|         ( ( ((char*)(fullFileName->Buffer))[1] ) ==
    | '?' )
46526|     ) {
46527|         Debug(DEBUG_FILE,("Invalid name\n"));
46528|     }
46529| #endif
46530|
46531|     return fullFileName;
46532| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
46533|     // Some kind of fault (probably a Page Fault
    | from a bad pointer)
46534|     // do cleanup...
46535|
46536|     Debug(DEBUG_FILE,("Error! Fault %08x in
    | File_GetFullFileName\n",GetExceptionCode()));
46537|
46538|     if(fullFileName) {
46539|         FREE_POINTER(fullFileName);
46540|     }
46541|
46542|     fullFileName = GetUniString( 3*sizeof(WCHAR) );
46543|     if(fullFileName) {
46544|         wcscpy( fullFileName->Buffer, L"???" );
46545|     }
46546|     return fullFileName;
46547| }
46548| }
46549| /*lint -restore */

```

```

46550|
46551|
46552| /*-----*/
46553| |-----*/
46553| int File_IsPagingFile (
46554|         PDEVICE_OBJECT DeviceObject,
46555|         PIRP Irp
46556|         )
46557| {
46558|     int PagingFile=0;
46559|     PUNICODE_STRING FileName;
46560|     #define PAGEFILENAME L"pagefile.sys"
46561|     NOT_REFERENCED(DeviceObject);
46562|
46563|     if( ( Irp->Flags & IRP_PAGING_IO ) ||
46564|         ( Irp->Flags & IRP_SYNCHRONOUS_PAGING_IO ) ) {
46565|         FileName = File_GetFileName(
46566|             | Irp->Tail.Overlay.OriginalFileObject );
46567|         if (FileName) {
46568|             if(FileName->Length >= 24) {
46569|                 WCHAR *p = FileName->Buffer +
46570|                     | (FileName->Length / sizeof(WCHAR)) - 12;
46571|                 if(_wcsicmp(p,PAGEFILENAME)==0) {
46572|                     PagingFile = 1;
46573|                 }
46574|             }
46575|             FREE_POINTER(FileName);
46576|         }
46577|
46578|         // will this ever happen???
46579|         if ( (!PagingFile) && (Irp->Flags &
46580|             | IRP_ASSOCIATED_IRP) ) {
46581|             if( ( Irp->AssociatedIrp.MasterIrp->Flags &
46582|                 | IRP_PAGING_IO ) ||
46583|                 ( Irp->AssociatedIrp.MasterIrp->Flags &
46584|                 | IRP_SYNCHRONOUS_PAGING_IO ) ) {
46585|                 FileName = File_GetFileName(
46586|                     | Irp->AssociatedIrp.MasterIrp->Tail.Overlay.OriginalFileO
46587|                     | bject );
46588|                 if(FileName) {
46589|                     if(FileName->Length >= 24) {
46590|                         WCHAR *p = FileName->Buffer +
46591|                             | (FileName->Length / sizeof(WCHAR)) - 12;
46592|                         if(_wcsicmp(p,PAGEFILENAME)==0) {
46593|                             PagingFile = 1;
46594|                         }
46595|                     }
46596|                 }
46597|             }
46598|         }
46599|     }
46600| }

```

```

46591|         FREE_POINTER(FileName);
46592|     }
46593| }
46594| }
46595|
46596| return PagingFile;
46597| }
46598|
46599| #ifndef _NTIFS_
46600| #if _WIN32_WINNT<0x0500
46601| typedef struct _FILE_NAME_INFORMATION {
46602|     ULONG FileNameLength;
46603|     WCHAR FileName[1];
46604| } FILE_NAME_INFORMATION, *PFILE_NAME_INFORMATION;
46605| #endif
46606|
46607| extern "C"
46608| NTKERNELAPI
46609| NTSTATUS
46610| IoQueryFileInformation(
46611|     IN PFILE_OBJECT FileObject,
46612|     IN FILE_INFORMATION_CLASS FileInformationClass,
46613|     IN ULONG Length,
46614|     OUT PVOID FileInformation,
46615|     OUT PULONG ReturnedLength
46616| );
46617| #endif
46618|
46619|
46620|
46621| #ifdef DEBUG
46622| /*-----*/
46623| | -----*/
46624| void File_PrintIrp(
46625|     PCCHAR Msg,
46626|     PDEVICE_OBJECT DeviceObject,
46627|     PIRP Irp
46628| ) {
46629|     PIO_STACK_LOCATION currentIrpStack =
46630|         | IoGetCurrentIrpStackLocation(Irp);
46631|     PFILTERED_EXTENSION DevExt =
46632|         | GetFilteredExtension(DeviceObject);
46633|     PUNICODE_STRING FileName;
46634|     ULONG Remainder;
46635|     ULARGE_INTEGER UL;
46636|     ULONG Sector;
46637|     UL.QuadPart =
46638|         | currentIrpStack->Parameters.Write.ByteOffset.QuadPart;

```



```

46637|   Sector = RtlEnlargedUnsignedDivide ( UL, 512,
      | &Remainder);
46638|
46639|   FileName = File_GetFullFileName(
      | Irp->Tail.Overlay.OriginalFileObject );
46640|
46641|   Debug(DEBUG_WRITE,("%s DO %p (%S) Irp %p Mdl %p
      | Flags %08x Buff %p Mode %p\n",
46642|       Msg,
46643|       DeviceObject, DevExt->Name,
46644|       Irp,
46645|       Irp->MdlAddress,
46646|       Irp->Flags,
46647|       Irp->AssociatedIrp.SystemBuffer,
46648|       Irp->RequestorMode
46649|   ));
46650|
46651|   Debug(DEBUG_WRITE,(" Pnd %d Stk %d/%d Cncl %d Apc
      | %d Znd %d losb %p Evt %p CmpKey %08x\n",
46652|       Irp->PendingReturned,
46653|       Irp->CurrentLocation,
46654|       Irp->StackCount,
46655|       Irp->Cancel,
46656|       Irp->ApcEnvironment,
46657|       Irp->AllocationFlags,
46658|       Irp->Userlosb,
46659|       Irp->UserEvent,
46660|       Irp->Tail.CompletionKey
46661|   ));
46662|
46663|   Debug(DEBUG_WRITE,(" CanRtn %p UserB %p Thd %p
      | AuxBuf %p OFO %p %wZ\n",
46664|       Irp->CancelRoutine,
46665|       Irp->UserBuffer,
46666|       Irp->Tail.Overlay.Thread,
46667|       Irp->Tail.Overlay.AuxiliaryBuffer,
46668|       Irp->Tail.Overlay.OriginalFileObject,
46669|       FileName
46670|   ));
46671|
46672|   Debug(DEBUG_WRITE,(" Stack: %s %d.%d Flgs %02x Ctrl
      | %d Ofs %12d Len %3d\n",
46673|       | File_GetMajorFunctionName(currentIrpStack->MajorFunction
      | ),
46674|       currentIrpStack->MajorFunction,
46675|       currentIrpStack->MinorFunction,
46676|       currentIrpStack->Flags,
46677|       currentIrpStack->Control,
46678|       Sector,

```

```

46679|     currentIrpStack->Parameters.Write.Length / 512
46680| ));
46681|
46682| Debug(DEBUG_WRITE,("  Key %08x DO %p FO %p CmpRtn
    | %p Ctxt %p\n",
46683|     currentIrpStack->Parameters.Write.Key,
46684|     currentIrpStack->DeviceObject,
46685|     currentIrpStack->FileObject,
46686|     currentIrpStack->CompletionRoutine,
46687|     currentIrpStack->Context
46688| ));
46689|
46690| if (FileName) {
46691|     FREE_POINTER(FileName);
46692| }
46693|
46694| }
46695|
46696| /* converts a unicode string to ansi and returns back a
    | string
46697| */
46698| /*-----
    | -----*/
46699| void SimpleUniToAnsi( PUNICODE_STRING Unicode, char
    | *Ansi )
46700| {
46701|     int i;
46702|     for (i=0;i<Unicode->Length / 2;i++) {
46703|         Ansi[i] = (char)Unicode->Buffer[i];
46704|     }
46705|
46706|     Ansi[Unicode->Length / 2] = 0;
46707|     return;
46708| }
46709|
46710| /*-----
    | -----*/
46711| void File_PrintOneLiner(
46712|     PCCHAR Msg,
46713|     PDEVICE_OBJECT DeviceObject,
46714|     PIRP Irp
46715| )
46716| {
46717|     PIO_STACK_LOCATION currentIrpStack =
    | IoGetCurrentIrpStackLocation(Irp);
46718|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DeviceObject);
46719|     PUNICODE_STRING FileName;
46720|     ULONG Remainder;
46721|     ULARGE_INTEGER UL;

```

```

46722|   ULONG Sector;
46723|
46724|   UL.QuadPart =
      | currentIrpStack->Parameters.Write.ByteOffset.QuadPart;
46725|   Sector = RtlEnlargedUnsignedDivide ( UL, 512,
      | &Remainder);
46726|
46727|   //Debug(DEBUG_WRITE,("%08x %08x Getting name
      | (%08x)\n", Irp,Irp->Tail.Overlay.OriginalFileObject,KeGet
      | CurrentIrql()));
46728|   FileName = File_GetFullFileName(
      | Irp->Tail.Overlay.OriginalFileObject );
46729|   if(FileName) {
46730|       char *Buff=(char
      | *)MemAllocatePoolWithTag(NonPagedPool,FileName->Length+1
      | ,FILENAMETAG);
46731|       if(Buff) {
46732|           // we cant convert unicode chars at
      | Irql>=DISPATCH_LEVEL
46733|           SimpleUniToAnsi(FileName,Buff);
46734|
46735|           if(KeGetCurrentIrql()<DISPATCH_LEVEL) {
46736|               Debug(DEBUG_PSMFILES,( "%08x %-25.25s
      | '%S' %08x %3x '%s'\n",Irp,
46737|               Msg, DevExt->Name,
46738|               Sector,
46739|
      | currentIrpStack->Parameters.Write.Length / 512,
46740|               Buff
46741|               ));
46742|           } else {
46743|               Debug(DEBUG_PSMFILES,( "%08x %-25.25s
      | '(At Dispatch Level)' %08x %3x '%s'\n",Irp,
46744|               Msg,
46745|               Sector,
46746|
      | currentIrpStack->Parameters.Write.Length / 512,
46747|               Buff
46748|               ));
46749|           }
46750|           FREE_POINTER(Buff);
46751|       }
46752|
46753|       FREE_POINTER(FileName);
46754|   }
46755| }
46756|
46757|
46758| /*-----
      | -----*/

```

```

46759| char *File_GetIOCTLString( ULONG IoControlCode )
46760| {
46761|     switch(IoControlCode) {
46762|
46763|         // Disk Class driver
46764|         case IOCTL_DISK_GET_DRIVE_GEOMETRY : return
46765|             | "IOCTL_DISK_GET_DRIVE_GEOMETRY";
46766|         case IOCTL_DISK_GET_PARTITION_INFO : return
46767|             | "IOCTL_DISK_GET_PARTITION_INFO";
46768|         case IOCTL_DISK_SET_PARTITION_INFO : return
46769|             | "IOCTL_DISK_SET_PARTITION_INFO";
46770|         case IOCTL_DISK_GET_DRIVE_LAYOUT : return
46771|             | "IOCTL_DISK_GET_DRIVE_LAYOUT";
46772|         case IOCTL_DISK_SET_DRIVE_LAYOUT : return
46773|             | "IOCTL_DISK_SET_DRIVE_LAYOUT";
46774|         case IOCTL_DISK_VERIFY : return
46775|             | "IOCTL_DISK_VERIFY";
46776|         case IOCTL_DISK_FORMAT_TRACKS : return
46777|             | "IOCTL_DISK_FORMAT_TRACKS";
46778|         case IOCTL_DISK_REASSIGN_BLOCKS : return
46779|             | "IOCTL_DISK_REASSIGN_BLOCKS";
46780|         case IOCTL_DISK_PERFORMANCE : return
46781|             | "IOCTL_DISK_PERFORMANCE";
46782|         case IOCTL_DISK_IS_WRITABLE : return
46783|             | "IOCTL_DISK_IS_WRITABLE";
46784|         case IOCTL_DISK_LOGGING : return
46785|             | "IOCTL_DISK_LOGGING";
46786|         case IOCTL_DISK_FORMAT_TRACKS_EX : return
46787|             | "IOCTL_DISK_FORMAT_TRACKS_EX";
46788|         case IOCTL_DISK_HISTOGRAM_STRUCTURE : return
46789|             | "IOCTL_DISK_HISTOGRAM_STRUCTURE";
46790|         case IOCTL_DISK_HISTOGRAM_DATA : return
46791|             | "IOCTL_DISK_HISTOGRAM_DATA";
46792|         case IOCTL_DISK_HISTOGRAM_RESET : return
46793|             | "IOCTL_DISK_HISTOGRAM_RESET";
46794|         case IOCTL_DISK_REQUEST_STRUCTURE : return
46795|             | "IOCTL_DISK_REQUEST_STRUCTURE";
46796|         case IOCTL_DISK_REQUEST_DATA : return
46797|             | "IOCTL_DISK_REQUEST_DATA";
46798|
46799|
46800|
46801|         #if(_WIN32_WINNT >= 0x0400)
46802|         case IOCTL_DISK_CONTROLLER_NUMBER : return
46803|             | "IOCTL_DISK_CONTROLLER_NUMBER";
46804|         case SMART_GET_VERSION : return
46805|             | "SMART_GET_VERSION";
46806|         case SMART_SEND_DRIVE_COMMAND : return
46807|             | "SMART_SEND_DRIVE_COMMAND";
46808|         case SMART_RCV_DRIVE_DATA : return
46809|             | "SMART_RCV_DRIVE_DATA";

```

```

46788| #else
46789|     case IOCTL_DISK_REMOVE_DEVICE      : return
      | "IOCTL_DISK_REMOVE_DEVICE";
46790| #endif /* _WIN32_WINNT >= 0x0400 */
46791|
46792|     // internal
46793|     case IOCTL_DISK_INTERNAL_SET_VERIFY : return
      | "IOCTL_DISK_INTERNAL_SET_VERIFY";
46794|     case IOCTL_DISK_INTERNAL_CLEAR_VERIFY : return
      | "IOCTL_DISK_INTERNAL_CLEAR_VERIFY";
46795|
46796|     // common to all class drivers
46797|     case IOCTL_DISK_CHECK_VERIFY      : return
      | "IOCTL_DISK_CHECK_VERIFY";
46798|     case IOCTL_DISK_MEDIA_REMOVAL     : return
      | "IOCTL_DISK_MEDIA_REMOVAL";
46799|     case IOCTL_DISK_EJECT_MEDIA       : return
      | "IOCTL_DISK_EJECT_MEDIA";
46800|     case IOCTL_DISK_LOAD_MEDIA        : return
      | "IOCTL_DISK_LOAD_MEDIA";
46801|     case IOCTL_DISK_RESERVE           : return
      | "IOCTL_DISK_RESERVE";
46802|     case IOCTL_DISK_RELEASE           : return
      | "IOCTL_DISK_RELEASE";
46803|     case IOCTL_DISK_FIND_NEW_DEVICES  : return
      | "IOCTL_DISK_FIND_NEW_DEVICES";
46804|
46805| #if _WIN32_WINNT >= 0x0500
46806|     case IOCTL_DISK_UPDATE_DRIVE_SIZE : return
      | "IOCTL_DISK_UPDATE_DRIVE_SIZE";
46807|     case IOCTL_DISK_GROW_PARTITION    : return
      | "IOCTL_DISK_GROW_PARTITION";
46808|     case IOCTL_DISK_GET_CACHE_INFORMATION : return
      | "IOCTL_DISK_GET_CACHE_INFORMATION";
46809|     case IOCTL_DISK_SET_CACHE_INFORMATION : return
      | "IOCTL_DISK_SET_CACHE_INFORMATION";
46810|     case IOCTL_DISK_DELETE_DRIVE_LAYOUT : return
      | "IOCTL_DISK_DELETE_DRIVE_LAYOUT";
46811|     case IOCTL_DISK_FORMAT_DRIVE      : return
      | "IOCTL_DISK_FORMAT_DRIVE";
46812|     case IOCTL_DISK_SENSE_DEVICE      : return
      | "IOCTL_DISK_SENSE_DEVICE";
46813|     case IOCTL_DISK_INTERNAL_SET_NOTIFY : return
      | "IOCTL_DISK_INTERNAL_SET_NOTIFY";
46814| #endif
46815|
46816|     case IOCTL_DISK_SIMBAD            : return
      | "IOCTL_DISK_SIMBAD";
46817|
46818|     case IOCTL_STORAGE_CHECK_VERIFY   : return

```

```

    | "IOCTL_STORAGE_CHECK_VERIFY";
46819|     case IOCTL_STORAGE_MEDIA_REMOVAL      : return
    | "IOCTL_STORAGE_MEDIA_REMOVAL";
46820|     case IOCTL_STORAGE_EJECT_MEDIA        : return
    | "IOCTL_STORAGE_EJECT_MEDIA";
46821|     case IOCTL_STORAGE_LOAD_MEDIA         : return
    | "IOCTL_STORAGE_LOAD_MEDIA";
46822|     case IOCTL_STORAGE_RESERVE            : return
    | "IOCTL_STORAGE_RESERVE";
46823|     case IOCTL_STORAGE_RELEASE            : return
    | "IOCTL_STORAGE_RELEASE";
46824|     case IOCTL_STORAGE_FIND_NEW_DEVICES   : return
    | "IOCTL_STORAGE_FIND_NEW_DEVICES";
46825|
46826| #if _WIN32_WINNT >= 0x0500
46827|     case IOCTL_STORAGE_CHECK_VERIFY2      : return
    | "IOCTL_STORAGE_CHECK_VERIFY2";
46828|     case IOCTL_STORAGE_LOAD_MEDIA2        : return
    | "IOCTL_STORAGE_LOAD_MEDIA2";
46829|     case IOCTL_STORAGE_EJECTION_CONTROL    : return
    | "IOCTL_STORAGE_EJECTION_CONTROL";
46830|     case IOCTL_STORAGE_MCN_CONTROL         : return
    | "IOCTL_STORAGE_MCN_CONTROL";
46831|     case IOCTL_STORAGE_GET_MEDIA_TYPES_EX : return
    | "IOCTL_STORAGE_GET_MEDIA_TYPES_EX";
46832|     case IOCTL_STORAGE_RESET_BUS          : return
    | "IOCTL_STORAGE_RESET_BUS";
46833|     case IOCTL_STORAGE_RESET_DEVICE        : return
    | "IOCTL_STORAGE_RESET_DEVICE";
46834|     case IOCTL_STORAGE_GET_DEVICE_NUMBER   : return
    | "IOCTL_STORAGE_GET_DEVICE_NUMBER";
46835|     case IOCTL_STORAGE_PREDICT_FAILURE     : return
    | "IOCTL_STORAGE_PREDICT_FAILURE";
46836|     case IOCTL_STORAGE_QUERY_PROPERTY      : return
    | "IOCTL_STORAGE_QUERY_PROPERTY";
46837| #endif
46838|
46839|     // SCSI Class driver
46840|     case IOCTL_DISK_GET_MEDIA_TYPES       : return
    | "IOCTL_DISK_GET_MEDIA_TYPES";
46841|     case IOCTL SCSI_PASS_THROUGH          : return
    | "IOCTL SCSI_PASS_THROUGH";
46842|     case IOCTL SCSI_MINIPORT              : return
    | "IOCTL SCSI_MINIPORT";
46843|     case IOCTL SCSI_GET_INQUIRY_DATA       : return
    | "IOCTL SCSI_GET_INQUIRY_DATA";
46844|     case IOCTL SCSI_GET_CAPABILITIES       : return
    | "IOCTL SCSI_GET_CAPABILITIES";
46845|     case IOCTL SCSI_PASS_THROUGH_DIRECT    : return
    | "IOCTL SCSI_PASS_THROUGH_DIRECT";

```

```

46846|     case IOCTL_SCSI_GET_ADDRESS      : return
      | "IOCTL_SCSI_GET_ADDRESS";
46847|     case IOCTL_SCSI_RESCAN_BUS        : return
      | "IOCTL_SCSI_RESCAN_BUS";
46848|     case IOCTL_SCSI_GET_DUMP_POINTERS  : return
      | "IOCTL_SCSI_GET_DUMP_POINTERS";
46849|
46850| #if _WIN32_WINNT >=0x0500
46851|     case IOCTL_SCSI_FREE_DUMP_POINTERS : return
      | "IOCTL_SCSI_FREE_DUMP_POINTERS";
46852|     case IOCTL_IDE_PASS_THROUGH        : return
      | "IOCTL_IDE_PASS_THROUGH";
46853| #endif
46854|
46855|     // fault tolerant driver
46856|     case FT_INITIALIZE_SET              : return
      | "FT_INITIALIZE_SET";
46857|     case FT_REGENERATE                  : return
      | "FT_REGENERATE";
46858|     case FT_CONFIGURE                   : return
      | "FT_CONFIGURE";
46859|     case FT_VERIFY                      : return
      | "FT_VERIFY";
46860|     case FT_SECONDARY_READ              : return
      | "FT_SECONDARY_READ";
46861|     case FT_PRIMARY_READ                : return
      | "FT_PRIMARY_READ";
46862|     case FT_BALANCED_READ_MODE          : return
      | "FT_BALANCED_READ_MODE";
46863|     case FT_SYNC_REDUNDANT_COPY         : return
      | "FT_SYNC_REDUNDANT_COPY";
46864|     case FT_SEQUENTIAL_WRITE_MODE       : return
      | "FT_SEQUENTIAL_WRITE_MODE";
46865|     case FT_PARALLEL_WRITE_MODE         : return
      | "FT_PARALLEL_WRITE_MODE";
46866|     case FT_QUERY_SET_STATE             : return
      | "FT_QUERY_SET_STATE";
46867|
46868| #if _WIN32_WINNT >=0x0500
46869|     case FT_CLUSTER_SET_MEMBER_STATE    : return
      | "FT_CLUSTER_SET_MEMBER_STATE";
46870|     case FT_CLUSTER_GET_MEMBER_STATE    : return
      | "FT_CLUSTER_GET_MEMBER_STATE";
46871|
46872|     // fault tolereant driver 2
46873|     case FT_CREATE_LOGICAL_DISK
      | : return "FT_CREATE_LOGICAL_DISK";
46874|     case FT_BREAK_LOGICAL_DISK
      | : return "FT_BREAK_LOGICAL_DISK";
46875|     case FT_ENUMERATE_LOGICAL_DISKS

```

```

| : return "FT_ENUMERATE_LOGICAL_DISKS";
46876|     case FT_QUERY_LOGICAL_DISK_INFORMATION
| : return "FT_QUERY_LOGICAL_DISK_INFORMATION";
46877|     case FT_ORPHAN_LOGICAL_DISK_MEMBER
| : return "FT_ORPHAN_LOGICAL_DISK_MEMBER";
46878|     case FT_REPLACE_LOGICAL_DISK_MEMBER
| : return "FT_REPLACE_LOGICAL_DISK_MEMBER";
46879|     case FT_QUERY_NT_DEVICE_NAME_FOR_LOGICAL_DISK
| : return "FT_QUERY_NT_DEVICE_NAME_FOR_LOGICAL_DISK";
46880|     case FT_INITIALIZE_LOGICAL_DISK
| : return "FT_INITIALIZE_LOGICAL_DISK";
46881|     case FT_QUERY_DRIVE_LETTER_FOR_LOGICAL_DISK
| : return "FT_QUERY_DRIVE_LETTER_FOR_LOGICAL_DISK";
46882|     case FT_CHECK_IO
| : return "FT_CHECK_IO";
46883|     case FT_SET_DRIVE_LETTER_FOR_LOGICAL_DISK
| : return "FT_SET_DRIVE_LETTER_FOR_LOGICAL_DISK";
46884|     case FT_QUERY_NT_DEVICE_NAME_FOR_PARTITION
| : return "FT_QUERY_NT_DEVICE_NAME_FOR_PARTITION";
46885|     case FT_CHANGE_NOTIFY
| : return "FT_CHANGE_NOTIFY";
46886|     case FT_STOP_SYNC_OPERATIONS
| : return "FT_STOP_SYNC_OPERATIONS";
46887|     case FT_QUERY_LOGICAL_DISK_ID
| : return "FT_QUERY_LOGICAL_DISK_ID";
46888|     case FT_CREATE_PARTITION_LOGICAL_DISK
| : return "FT_CREATE_PARTITION_LOGICAL_DISK";
46889|
46890|     // volume manager
46891|     case IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS :
| return "IOCTL_VOLUME_GET_VOLUME_DISK_EXTENTS";
46892|     case IOCTL_VOLUME_SUPPORTS_ONLINE_OFFLINE :
| return "IOCTL_VOLUME_SUPPORTS_ONLINE_OFFLINE";
46893|     case IOCTL_VOLUME_ONLINE :
| return "IOCTL_VOLUME_ONLINE";
46894|     case IOCTL_VOLUME_OFFLINE :
| return "IOCTL_VOLUME_OFFLINE";
46895|     case IOCTL_VOLUME_IS_OFFLINE :
| return "IOCTL_VOLUME_IS_OFFLINE";
46896|     case IOCTL_VOLUME_IS_IO_CAPABLE :
| return "IOCTL_VOLUME_IS_IO_CAPABLE";
46897|     case IOCTL_VOLUME_QUERY_FAILOVER_SET :
| return "IOCTL_VOLUME_QUERY_FAILOVER_SET";
46898|     case IOCTL_VOLUME_QUERY_VOLUME_NUMBER :
| return "IOCTL_VOLUME_QUERY_VOLUME_NUMBER";
46899|     case IOCTL_VOLUME_LOGICAL_TO_PHYSICAL :
| return "IOCTL_VOLUME_LOGICAL_TO_PHYSICAL";
46900|     case IOCTL_VOLUME_PHYSICAL_TO_LOGICAL :
| return "IOCTL_VOLUME_PHYSICAL_TO_LOGICAL";
46901|

```



```

46902|    // mountmgr
46903|    case IOCTL_MOUNTMGR_CREATE_POINT
    | : return "IOCTL_MOUNTMGR_CREATE_POINT";
46904|    case IOCTL_MOUNTMGR_DELETE_POINTS
    | : return "IOCTL_MOUNTMGR_DELETE_POINTS";
46905|    case IOCTL_MOUNTMGR_QUERY_POINTS
    | : return "IOCTL_MOUNTMGR_QUERY_POINTS";
46906|    case IOCTL_MOUNTMGR_DELETE_POINTS_DBONLY
    | : return "IOCTL_MOUNTMGR_DELETE_POINTS_DBONLY";
46907|    case IOCTL_MOUNTMGR_NEXT_DRIVE_LETTER
    | : return "IOCTL_MOUNTMGR_NEXT_DRIVE_LETTER";
46908|    case IOCTL_MOUNTMGR_AUTO_DL_ASSIGNMENTS
    | : return "IOCTL_MOUNTMGR_AUTO_DL_ASSIGNMENTS";
46909|    case IOCTL_MOUNTMGR_VOLUME_MOUNT_POINT_CREATED
    | : return "IOCTL_MOUNTMGR_VOLUME_MOUNT_POINT_CREATED";
46910|    case IOCTL_MOUNTMGR_VOLUME_MOUNT_POINT_DELETED
    | : return "IOCTL_MOUNTMGR_VOLUME_MOUNT_POINT_DELETED";
46911|    case IOCTL_MOUNTMGR_CHANGE_NOTIFY
    | : return "IOCTL_MOUNTMGR_CHANGE_NOTIFY";
46912|    case IOCTL_MOUNTMGR_KEEP_LINKS_WHEN_OFFLINE
    | : return "IOCTL_MOUNTMGR_KEEP_LINKS_WHEN_OFFLINE";
46913|    case IOCTL_MOUNTMGR_CHECK_UNPROCESSED_VOLUMES
    | : return "IOCTL_MOUNTMGR_CHECK_UNPROCESSED_VOLUMES";
46914|    case IOCTL_MOUNTMGR_VOLUME_ARRIVAL_NOTIFICATION
    | : return "IOCTL_MOUNTMGR_VOLUME_ARRIVAL_NOTIFICATION";
46915|
46916|    // supported on all mounted devices
46917|    case IOCTL_MOUNTDEV_QUERY_UNIQUE_ID
    | : return "IOCTL_MOUNTDEV_QUERY_UNIQUE_ID";
46918|    case IOCTL_MOUNTDEV_QUERY_DEVICE_NAME
    | : return "IOCTL_MOUNTDEV_QUERY_DEVICE_NAME";
46919|    case IOCTL_MOUNTDEV_UNIQUE_ID_CHANGE_NOTIFY
    | : return "IOCTL_MOUNTDEV_UNIQUE_ID_CHANGE_NOTIFY";
46920|    case IOCTL_MOUNTDEV_QUERY_SUGGESTED_LINK_NAME
    | : return "IOCTL_MOUNTDEV_QUERY_SUGGESTED_LINK_NAME";
46921|    case IOCTL_MOUNTDEV_LINK_CREATED
    | : return "IOCTL_MOUNTDEV_LINK_CREATED";
46922|    case IOCTL_MOUNTDEV_LINK_DELETED
    | : return "IOCTL_MOUNTDEV_LINK_DELETED";
46923| #ifndef IOCTL_MOUNTDEV_QUERY_STABLE_GUID
46924| #define IOCTL_MOUNTDEV_QUERY_STABLE_GUID
    | CTL_CODE(MOUNTDEVCONTROLTYPE, 6, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
46925| #endif
46926|
46927|    // added in whistler build 2416
46928|    case IOCTL_MOUNTDEV_QUERY_STABLE_GUID : return
    | "IOCTL_MOUNTDEV_QUERY_STABLE_GUID";
46929|
46930|    // new disk ioctls from whistler build 2416

```

```

46931|     case CTL_CODE(IOCTL_DISK_BASE, 0x0012,
| METHOD_BUFFERED, FILE_READ_ACCESS):
46932|     case IOCTL_DISK_GET_PARTITION_INFO_EX :
| return "IOCTL_DISK_GET_PARTITION_INFO_EX";
46933|     case IOCTL_DISK_SET_PARTITION_INFO_EX :
| return "IOCTL_DISK_SET_PARTITION_INFO_EX";
46934|     case CTL_CODE(IOCTL_DISK_BASE, 0x0014,
| METHOD_BUFFERED, FILE_READ_ACCESS):
46935|     case IOCTL_DISK_GET_DRIVE_LAYOUT_EX :
| return "IOCTL_DISK_GET_DRIVE_LAYOUT_EX";
46936|     case IOCTL_DISK_SET_DRIVE_LAYOUT_EX :
| return "IOCTL_DISK_SET_DRIVE_LAYOUT_EX";
46937|     case IOCTL_DISK_CREATE_DISK :
| return "IOCTL_DISK_CREATE_DISK";
46938|     case IOCTL_DISK_GET_LENGTH_INFO :
| return "IOCTL_DISK_GET_LENGTH_INFO";
46939|     case CTL_CODE(IOCTL_DISK_BASE, 0x0028,
| METHOD_BUFFERED, FILE_READ_ACCESS):
46940|     case IOCTL_DISK_GET_DRIVE_GEOMETRY_EX :
| return "IOCTL_DISK_GET_DRIVE_GEOMETRY_EX";
46941|
46942| #ifndef IOCTL_ACPI_ASYNC_EVAL_METHOD
46943| #define IOCTL_ACPI_ASYNC_EVAL_METHOD
| CTL_CODE(FILE_DEVICE_ACPI, 0, METHOD_BUFFERED,
| FILE_READ_ACCESS | FILE_WRITE_ACCESS)
46944| #define IOCTL_ACPI_EVAL_METHOD
| CTL_CODE(FILE_DEVICE_ACPI, 1, METHOD_BUFFERED,
| FILE_READ_ACCESS | FILE_WRITE_ACCESS)
46945| #define IOCTL_ACPI_ACQUIRE_GLOBAL_LOCK
| CTL_CODE(FILE_DEVICE_ACPI, 4, METHOD_BUFFERED,
| FILE_READ_ACCESS | FILE_WRITE_ACCESS)
46946| #define IOCTL_ACPI_RELEASE_GLOBAL_LOCK
| CTL_CODE(FILE_DEVICE_ACPI, 5, METHOD_BUFFERED,
| FILE_READ_ACCESS | FILE_WRITE_ACCESS)
46947| #endif
46948|     // whistler build 2416 sends down acpi commands
| to the disk driver for some reason..
46949|     case IOCTL_ACPI_ASYNC_EVAL_METHOD : return
| "IOCTL_ACPI_ASYNC_EVAL_METHOD";
46950|     case IOCTL_ACPI_EVAL_METHOD : return
| "IOCTL_ACPI_EVAL_METHOD";
46951|     case IOCTL_ACPI_ACQUIRE_GLOBAL_LOCK : return
| "IOCTL_ACPI_ACQUIRE_GLOBAL_LOCK";
46952|     case IOCTL_ACPI_RELEASE_GLOBAL_LOCK : return
| "IOCTL_ACPI_RELEASE_GLOBAL_LOCK";
46953|
46954|
46955| #endif
46956|
46957|     default:

```

```

46958|     return "Unknown IOCTL code";
46959| }
46960| }
46961|
46962| /*-----
| -----*/
46963| char *File_GetMajorFunctionName ( ULONG MajorFunction )
46964| {
46965|     switch (MajorFunction) {
46966|         case IRP_MJ_CREATE                :
| return "IRP_MJ_CREATE";
46967|         case IRP_MJ_CREATE_NAMED_PIPE    :
| return "IRP_MJ_CREATE_NAMED_PIPE";
46968|         case IRP_MJ_CLOSE                :
| return "IRP_MJ_CLOSE";
46969|         case IRP_MJ_READ                  :
| return "IRP_MJ_READ";
46970|         case IRP_MJ_WRITE                 :
| return "IRP_MJ_WRITE";
46971|         case IRP_MJ_QUERY_INFORMATION     :
| return "IRP_MJ_QUERY_INFORMATION";
46972|         case IRP_MJ_SET_INFORMATION       :
| return "IRP_MJ_SET_INFORMATION";
46973|         case IRP_MJ_QUERY_EA              :
| return "IRP_MJ_QUERY_EA";
46974|         case IRP_MJ_SET_EA                :
| return "IRP_MJ_SET_EA";
46975|         case IRP_MJ_FLUSH_BUFFERS         :
| return "IRP_MJ_FLUSH_BUFFERS";
46976|         case IRP_MJ_QUERY_VOLUME_INFORMATION :
| return "IRP_MJ_QUERY_VOLUME_INFORMATION";
46977|         case IRP_MJ_SET_VOLUME_INFORMATION :
| return "IRP_MJ_SET_VOLUME_INFORMATION";
46978|         case IRP_MJ_DIRECTORY_CONTROL     :
| return "IRP_MJ_DIRECTORY_CONTROL";
46979|         case IRP_MJ_FILE_SYSTEM_CONTROL   :
| return "IRP_MJ_FILE_SYSTEM_CONTROL";
46980|         case IRP_MJ_DEVICE_CONTROL        :
| return "IRP_MJ_DEVICE_CONTROL";
46981|         case IRP_MJ_INTERNAL_DEVICE_CONTROL :
| return "IRP_MJ_INTERNAL_DEVICE_CONTROL";
46982|         case IRP_MJ_SHUTDOWN              :
| return "IRP_MJ_SHUTDOWN";
46983|         case IRP_MJ_LOCK_CONTROL           :
| return "IRP_MJ_LOCK_CONTROL";
46984|         case IRP_MJ_CLEANUP               :
| return "IRP_MJ_CLEANUP";
46985|         case IRP_MJ_CREATE_MAILSLLOT      :
| return "IRP_MJ_CREATE_MAILSLLOT";
46986|         case IRP_MJ_QUERY_SECURITY        :

```

```

    | return "IRP_MJ_QUERY_SECURITY";
46987|     case IRP_MJ_SET_SECURITY          :
    | return "IRP_MJ_SET_SECURITY";
46988|
46989| #if _WIN32_WINNT<0x0500
46990|     case IRP_MJ_SET_POWER              :
    | return "IRP_MJ_SET_POWER";
46991|     case IRP_MJ_QUERY_POWER            :
    | return "IRP_MJ_QUERY_POWER";
46992| #else
46993|     case IRP_MJ_POWER                  :
    | return "IRP_MJ_POWER";
46994|     case IRP_MJ_SYSTEM_CONTROL         :
    | return "IRP_MJ_SYSTEM_CONTROL";
46995|     case IRP_MJ_DEVICE_CHANGE          :
    | return "IRP_MJ_DEVICE_CHANGE";
46996|     case IRP_MJ_QUERY_QUOTA            :
    | return "IRP_MJ_QUERY_QUOTA";
46997|     case IRP_MJ_SET_QUOTA              :
    | return "IRP_MJ_SET_QUOTA";
46998|     case IRP_MJ_PNP                   :
    | return "IRP_MJ_PNP";
46999| #endif
47000| default:
47001|     return "Unknown function";
47002| }
47003| }
47004|
47005|
47006| char *File_GetPnpMinorFunctionName( ULONG MinorFunction
    | )
47007| {
47008| #if _WIN32_WINNT>=0x0500
47009|     switch(MinorFunction) {
47010|         case IRP_MN_START_DEVICE        :
    | return "IRP_MN_START_DEVICE";
47011|         case IRP_MN_QUERY_REMOVE_DEVICE :
    | return "IRP_MN_QUERY_REMOVE_DEVICE";
47012|         case IRP_MN_REMOVE_DEVICE       :
    | return "IRP_MN_REMOVE_DEVICE";
47013|         case IRP_MN_CANCEL_REMOVE_DEVICE :
    | return "IRP_MN_CANCEL_REMOVE_DEVICE";
47014|         case IRP_MN_STOP_DEVICE          :
    | return "IRP_MN_STOP_DEVICE";
47015|         case IRP_MN_QUERY_STOP_DEVICE    :
    | return "IRP_MN_QUERY_STOP_DEVICE";
47016|         case IRP_MN_CANCEL_STOP_DEVICE   :
    | return "IRP_MN_CANCEL_STOP_DEVICE";
47017|         case IRP_MN_QUERY_DEVICE_RELATIONS :
    | return "IRP_MN_QUERY_DEVICE_RELATIONS";

```

```

47018|     case IRP_MN_QUERY_INTERFACE      :
      | return "IRP_MN_QUERY_INTERFACE";
47019|     case IRP_MN_QUERY_CAPABILITIES     :
      | return "IRP_MN_QUERY_CAPABILITIES";
47020|     case IRP_MN_QUERY_RESOURCES        :
      | return "IRP_MN_QUERY_RESOURCES";
47021|     case IRP_MN_QUERY_RESOURCE_REQUIREMENTS :
      | return "IRP_MN_QUERY_RESOURCE_REQUIREMENTS";
47022|     case IRP_MN_QUERY_DEVICE_TEXT      :
      | return "IRP_MN_QUERY_DEVICE_TEXT";
47023|     case IRP_MN_FILTER_RESOURCE_REQUIREMENTS:
      | return "IRP_MN_FILTER_RESOURCE_REQUIREMENTS";
47024|     case IRP_MN_READ_CONFIG            :
      | return "IRP_MN_READ_CONFIG";
47025|     case IRP_MN_WRITE_CONFIG           :
      | return "IRP_MN_WRITE_CONFIG";
47026|     case IRP_MN_EJECT                  :
      | return "IRP_MN_EJECT";
47027|     case IRP_MN_SET_LOCK                :
      | return "IRP_MN_SET_LOCK";
47028|     case IRP_MN_QUERY_ID                :
      | return "IRP_MN_QUERY_ID";
47029|     case IRP_MN_QUERY_PNP_DEVICE_STATE  :
      | return "IRP_MN_QUERY_PNP_DEVICE_STATE";
47030|     case IRP_MN_QUERY_BUS_INFORMATION   :
      | return "IRP_MN_QUERY_BUS_INFORMATION";
47031|     case IRP_MN_DEVICE_USAGE_NOTIFICATION :
      | return "IRP_MN_DEVICE_USAGE_NOTIFICATION";
47032|     case IRP_MN_SURPRISE_REMOVAL        :
      | return "IRP_MN_SURPRISE_REMOVAL";
47033|     case IRP_MN_QUERY_LEGACY_BUS_INFORMATION:
      | return "IRP_MN_QUERY_LEGACY_BUS_INFORMATION";
47034|     default:
47035|         return "Unknown function";
47036|     }
47037| #else
47038|     return "Unknown function";
47039| #endif
47040| }
47041|
47042| char *File_GetPowerMinorFunctionName( ULONG
      | MinorFunction )
47043| {
47044|     #if _WIN32_WINNT >= 0x0500
47045|     switch(MinorFunction) {
47046|         case IRP_MN_WAIT_WAKE      : return
          | "IRP_MN_WAIT_WAKE";
47047|         case IRP_MN_POWER_SEQUENCE : return
          | "IRP_MN_POWER_SEQUENCE";
47048|         case IRP_MN_SET_POWER      : return

```

```

    | "IRP_MN_SET_POWER";
47049|     case IRP_MN_QUERY_POWER    : return
    | "IRP_MN_QUERY_POWER";
47050|     default:
47051|         return "Unknown function";
47052|     }
47053| #else
47054|     return "Unknown function";
47055| #endif
47056| }
47057|
47058| char *File_GetSystemControlMinorFunctionName( ULONG
    | MinorFunction )
47059| {
47060| #if _WIN32_WINNT >= 0x0500
47061|     switch(MinorFunction) {
47062|         case IRP_MN_QUERY_ALL_DATA        : return
    | "IRP_MN_QUERY_ALL_DATA";
47063|         case IRP_MN_QUERY_SINGLE_INSTANCE : return
    | "IRP_MN_QUERY_SINGLE_INSTANCE";
47064|         case IRP_MN_CHANGE_SINGLE_INSTANCE : return
    | "IRP_MN_CHANGE_SINGLE_INSTANCE";
47065|         case IRP_MN_CHANGE_SINGLE_ITEM    : return
    | "IRP_MN_CHANGE_SINGLE_ITEM";
47066|         case IRP_MN_ENABLE_EVENTS         : return
    | "IRP_MN_ENABLE_EVENTS";
47067|         case IRP_MN_DISABLE_EVENTS        : return
    | "IRP_MN_DISABLE_EVENTS";
47068|         case IRP_MN_ENABLE_COLLECTION     : return
    | "IRP_MN_ENABLE_COLLECTION";
47069|         case IRP_MN_DISABLE_COLLECTION    : return
    | "IRP_MN_DISABLE_COLLECTION";
47070|         case IRP_MN_REGINFO               : return
    | "IRP_MN_REGINFO";
47071|         case IRP_MN_EXECUTE_METHOD        : return
    | "IRP_MN_EXECUTE_METHOD";
47072|         default:
47073|             return "Unknown function";
47074|     }
47075| #else
47076|     return "Unknown function";
47077| #endif
47078| }
47079|
47080| #endif
47081|
47082|
47083|
47084| File Listing: FILE.h
47085|

```

```

47086| PUNICODE_STRING File_GetFullFileName(
47087|         PFILE_OBJECT fileObject
47088|     );
47089|
47090| PUNICODE_STRING File_GetFileName(
47091|         PFILE_OBJECT fileObject
47092|     );
47093| int File_IsPagingFile (
47094|         PDEVICE_OBJECT DeviceObject,
47095|         PIRP Irp
47096|     );
47097|
47098| PUNICODE_STRING File_QueryFileName(
47099|         PFILE_OBJECT fileObject
47100|     );
47101| NTSTATUS File_QueryMFTEntryNumber(
47102|         PFILE_OBJECT fileObject,
47103|         ULARGE_INTEGER &Id);
47104|
47105| #define MAX_NTFS_ENTRY_NUM 16
47106| CHAR *NtFsMetaDataFileNames[];
47107|
47108| #ifdef DEBUG
47109| void File_PrintIrp(
47110|         PCCHAR Msg,
47111|         PDEVICE_OBJECT DeviceObject,
47112|         PIRP Irp
47113|     );
47114| void File_PrintOneLiner(
47115|         PCCHAR Msg,
47116|         PDEVICE_OBJECT DeviceObject,
47117|         PIRP Irp
47118|     );
47119|
47120| char *File_GetMajorFunctionName ( ULONG MajorFunction
    | );
47121| char *File_GetIoTypeName ( ULONG IoType );
47122| char *File_GetIoctlString( ULONG IoControlCode );
47123| char *File_GetPnpMinorFunctionName( ULONG MinorFunction
    | );
47124| char *File_GetSystemControlMinorFunctionName( ULONG
    | MinorFunction );
47125| char *File_GetPowerMinorFunctionName( ULONG
    | MinorFunction );
47126|
47127| #else
47128| #define File_PrintIrp( Msg, DeviceObject, Irp )
47129| #define File_PrintOneLiner( Msg, DeviceObject, Irp )
47130| #define File_GetMajorFunctionName ( MajorFunction )
47131| #define File_GetIoTypeName ( IoType )

```

```

47132| #define File_GetIOCTLString( IoControlCode )
47133| #define File_GetPnpMinorFunctionName( MinorFunction )
47134| #define File_GetSystemControlMinorFunctionName(
    | MinorFunction )
47135| #define File_GetPowerMinorFunctionName( MinorFunction )
47136| #endif
47137|
47138|
47139|
47140| File Listing: FLUSH.cpp
47141|
47142| #include "precomp.h"
47143|
47144|
47145| /*-----
    | -----*/
47146| NTSTATUS
47147| PSMANFlush(
47148|     IN PDEVICE_OBJECT DeviceObject,
47149|     IN PIRP Irp
47150| )
47151|
47152| /*++
47153|
47154| Routine Description:
47155|
47156|     Passes the Irp to the correct handler
47157|
47158| Arguments:
47159|
47160|     DriverObject - Pointer to device object to being
    | shutdown by system.
47161|     Irp          - IRP involved.
47162|
47163| Return Value:
47164|
47165|     NT Status
47166|
47167| --*/
47168|
47169| {
47170|
47171|     NTSTATUS Status;
47172|
47173|     switch(PsmGetObjectTypes(DeviceObject)) {
47174|         case OBJECT_INTERNAL :
47175|             Status = PSMANFlushObject(DeviceObject,
    | Irp);
47176|             break;
47177|         case OBJECT_FILTEREDDISK :

```



```

47178|         Status = PSMANFlushDevice(DeviceObject,
    | Irp);
47179|         break;
47180|     case OBJECT_VIRTUALDISK :
47181|         Status = PSMANFlushVdisk(DeviceObject,
    | Irp);
47182|         break;
47183|     case OBJECT_FS_OBJECT :
47184|         Status = PSMANFlushFSObject(DeviceObject,
    | Irp);
47185|         break;
47186|     case OBJECT_FS_FILTER :
47187|         Status = PSMANFlushFSFilter(DeviceObject,
    | Irp);
47188|         break;
47189|     default:
47190|         Irp->IoStatus.Status = Status =
    | STATUS_NO_SUCH_DEVICE;
47191|         Irp->IoStatus.Information = 0 ;
47192|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
47193|         break;
47194|     }
47195|     return Status;
47196|
47197| } // end PSMANFlush()
47198|
47199|
47200| /*-----
    | -----*/
47201| STATIC NTSTATUS
47202| PSMANFlushObject(
47203|     IN PDEVICE_OBJECT DeviceObject,
47204|     IN PIRP Irp
47205| )
47206|
47207| /*++
47208|
47209| Routine Description:
47210|
47211| This routine is called for a flush IRPs. These are
    | sent by the
47212| system when the file system does a flush.
47213|
47214| Arguments:
47215|
47216| DriverObject - Pointer to device object to being
    | shutdown by system.
47217| Irp - IRP involved. to being
    | shutdown by system.
47218|

```

```

47219| Return Value:
47220|
47221|     NT Status
47222|
47223| --*/
47224|
47225| {
47226|     NOT_REFERENCED(DeviceObject);
47227|     Debug(DEBUG_PROCCALL,("PSManFlushObject
    | Called\n"));
47228|     Irp->IoStatus.Status = STATUS_SUCCESS;
47229|     Irp->IoStatus.Information = 0;
47230|
47231|     IoCompleteRequest(Irp, IO_NO_INCREMENT);
47232|     Debug(DEBUG_PROCCALL,("PSManFlushObject Done\n"));
47233|     return STATUS_SUCCESS;
47234|
47235| } // end PSManFlushObject()
47236|
47237|
47238| /*-----
    | -----*/
47239| STATIC NTSTATUS
47240| PSManFlushDevice(
47241|     IN PDEVICE_OBJECT DeviceObject,
47242|     IN PIRP Irp
47243| )
47244|
47245| /*++
47246|
47247| Routine Description:
47248|
47249|     This routine is called for a flush IRPs. These are
    | sent by the
47250|     system when the file system does a flush.
47251|
47252| Arguments:
47253|
47254|     DriverObject - Pointer to device object to being
    | shutdown by system.
47255|     Irp          - IRP involved.
47256|
47257| Return Value:
47258|
47259|     NT Status
47260|
47261| --*/
47262|
47263| {
47264|     NTSTATUS Status;

```

```

47265| /*
47266|   PIO_STACK_LOCATION currentIrpStack =
47267|     | IoGetCurrentIrpStackLocation(Irp);
47268|   TRACE( TRACE_FLUSH,
47269|     | currentIrpStack->Parameters.Read.ByteOffset.HighPart,
47270|     | currentIrpStack->Parameters.Read.ByteOffset.LowPart,
47271|     | currentIrpStack->Parameters.Read.Length,
47272|     | currentIrpStack->Parameters.Read.Key,
47273|     | "");
47274| #ifdef DEBUG
47275|   if(PsmActive) {
47276|     Debug(DEBUG_FLUSH |
47277|       | DEBUG_PROCCALL,("PSManFlushDevice Called Device=%p,
47278|       | Irp=%p\n",DeviceObject,Irp));
47279|   }
47280| #endif
47281| */
47282|   Status = PSManPassThru( DeviceObject, Irp );
47283| /*
47284| #ifdef DEBUG
47285|   if(PsmActive) {
47286|     Debug(DEBUG_FLUSH |
47287|       | DEBUG_PROCCALL,("PSManFlushDevice Done Device=%p,
47288|       | Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
47289|   }
47290| #endif
47291| */
47292|   return Status;
47293| } // end PSManFlushDevice()
47294|
47295| /*-----*/
47296|
47297| STATIC NTSTATUS PSManFlushVDisk(
47298|   IN PDEVICE_OBJECT DeviceObject,
47299|   IN PIRP Irp
47300| )
47301| {
47302|   NTSTATUS Status=STATUS_SUCCESS;
47303|
47304|   Debug(DEBUG_PROCCALL | DEBUG_FLUSH,("PSManFlushVDisk
47305|     | Called Dev=%p, Irp=%p\n",DeviceObject,Irp));
47306|   Irp->IoStatus.Information = 0;
47307|   Irp->IoStatus.Status = Status;
47308|   IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
47309|   Debug(DEBUG_PROCCALL | DEBUG_FLUSH,("PSManFlushVDisk
47310|     | Done\n"));

```

```

47305|
47306|   return Status;
47307| }
47308|
47309|
47310| /*-----
    | -----*/
47311| STATIC NTSTATUS
47312| PSMANFlushFSFilter(
47313|   IN PDEVICE_OBJECT DeviceObject,
47314|   IN PIRP Irp
47315| )
47316|
47317| /*++
47318|
47319| Routine Description:
47320|
47321|   This routine is called for a flush IRPs. These are
    | sent by the
47322|   system when the file system does a flush.
47323|
47324| Arguments:
47325|
47326|   DriverObject - Pointer to device object to being
    | shutdown by system.
47327|   Irp          - IRP involved.
47328|
47329| Return Value:
47330|
47331|   NT Status
47332|
47333| --*/
47334|
47335| {
47336|   NTSTATUS Status;
47337| /*
47338| #ifdef DEBUG
47339|   if(PsmActive) {
47340|     Debug(DEBUG_FLUSH |
    | DEBUG_PROCCALL,("PSManFlushFSFilter Called Device=%p,
    | Irp=%p\n",DeviceObject,Irp));
47341|   }
47342| #endif
47343| */
47344|   Status = PSMANFSPassThru( DeviceObject, Irp );
47345| /*
47346| #ifdef DEBUG
47347|   if(PsmActive) {
47348|     Debug(DEBUG_FLUSH |
    | DEBUG_PROCCALL,("PSManFlushFSFilter Done   Device=%p,

```

```

    | Irp=%p, Status=%08x\n", DeviceObject, Irp, Status));
47349|    }
47350| #endif
47351| */
47352|    return Status;
47353|
47354| } // end PSMANFlushFSFilter()
47355|
47356|
47357| /*-----
    | -----*/
47358| STATIC NTSTATUS PSMANFlushFSObject(
47359|     IN PDEVICE_OBJECT DeviceObject,
47360|     IN PIRP Irp
47361| )
47362| {
47363|     NTSTATUS Status=STATUS_SUCCESS;
47364|
47365|     Debug(DEBUG_PROCCALL |
        | DEBUG_FLUSH,("PSMANFlushFSObject Called Dev=%p,
        | Irp=%p\n", DeviceObject, Irp));
47366|     Irp->IoStatus.Information = 0;
47367|     Irp->IoStatus.Status = Status;
47368|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
47369|     Debug(DEBUG_PROCCALL |
        | DEBUG_FLUSH,("PSMANFlushFSObject Done\n"));
47370|
47371|     return Status;
47372| }
47373|
47374|
47375|
47376| File Listing: FLUSH.h
47377|
47378| /*
47379| NTSTATUS
47380| PSMANFlush(
47381|     IN PDEVICE_OBJECT DeviceObject,
47382|     IN PIRP Irp
47383| );
47384|
47385| STATIC NTSTATUS
47386| PSMANFlushObject(
47387|     IN PDEVICE_OBJECT DeviceObject,
47388|     IN PIRP Irp
47389| );
47390|
47391| STATIC NTSTATUS
47392| PSMANFlushDevice(
47393|     IN PDEVICE_OBJECT DeviceObject,

```

```

47394| IN PIRP Irp
47395| );
47396|
47397| STATIC NTSTATUS PSMANFlushVDisk(
47398| IN PDEVICE_OBJECT DeviceObject,
47399| IN PIRP Irp
47400| );
47401|
47402| STATIC NTSTATUS PSMANFlushFSObject(
47403| IN PDEVICE_OBJECT DeviceObject,
47404| IN PIRP Irp
47405| );
47406|
47407| STATIC NTSTATUS PSMANFlushFSFilter(
47408| IN PDEVICE_OBJECT DeviceObject,
47409| IN PIRP Irp
47410| );
47411|
47412|
47413|
47414| File Listing: header.h
47415|
47416| #ifndef __PSM_HEADER_DATA_STRUCTURES
47417| #define __PSM_HEADER_DATA_STRUCTURES 3
47418|
47419| // a magic number which happens to be my birthday
47420| #define PSM_HEADER_SIGNATURE 0x03191972
47421| #define PSM_HEADER_MINOR_VERSION 0x10000000
47422| #define PSM_HEADER_MINOR_MASK 0xf0000000
47423| #define PSM_HEADER_SIGNATURE_MASK 0x0ffffff
47424|
47425| #define SIZE_OF_SNAPSHOT_BITMAP
47426| | (sizeof(RTL_BITMAP)+((MAX_NUMBER_OF_SNAPSHOTS+31) / 32)
47427| | * 4)
47428| typedef struct skSnapshotMaster *pkSnapshotMaster;
47429|
47430| typedef struct sDiskInternalSnapshot {
47431|     ULONG SequenceNumber;
47432|     LARGE_INTEGER SnapshotTime;
47433|     ULONG ExternalInstance;
47434|     ULONG GroupNumber;
47435|     ULONG Status;
47436|     ULONG NumToKeep;
47437|     BYTE Priority;
47438|     BYTE SnapshotFlags;
47439|     BYTE Reserved1;
47440|     BYTE Reserved2;
47441|     WCHAR UserSnapshotName[256];
47442| } tDiskInternalSnapshot, *pDiskInternalSnapshot;

```

```

47442| typedef struct sInternalSnapShot {
47443|     // The volatile, in-memory-only parts of a
    | snapshot...
47444|     LIST_ENTRY      ListEntry;
47445|     pkSnapShotMaster SnapShotMaster;
47446|     ULONG           ReferenceCount;
47447|     PVOID           DllPrivateUse;
47448|
47449|     // All the stuff that will be saved on disk
    | (but only if snapshot is persistent)...
47450|     tDiskInternalSnapShot Permanent;
47451| } tInternalSnapShot, *pInternalSnapShot;
47452|
47453|
47454|
    | //-----
    | -----
47455| // Data types needed for header file:
47456|
47457| #pragma warning (push)
47458| #pragma warning (disable:4200)
47459| /*lint -save -e43*/
47460| typedef struct sHeader {
47461|     ULONG           Version;
47462|     ULONG           Size;
47463|     ULONG           Signature;
47464|     ULONG           GranuleSizeInBytes;
47465|     ULONG           IndexLoadWheel;
47466|     ULONG           HighestSnapNumber;
47467|     LARGE_INTEGER   DateTimeWritten;
47468|     tRevertInfo      RevertInfo;
47469|     tDiskInternalSnapShot SnapShots[];
47470| } tHeader, *pHeader;
47471| /*lint -restore*/
47472| #pragma warning (pop)
47473|
47474| #define PSM_INDEX_FLAG_NORMAL          (0)
47475| #define PSM_INDEX_FLAG_VIRTUAL_WRITE  (1)
47476|
47477| typedef struct sDiskNode {
47478|     ULONG VolumeId;    // volume this granule
    | applies to
47479|     ULONG DasdGranule; // granule number of
    | data's original location on dasd
47480|     ULONG SnapshotNumber; // unique 32-bit
    | snapshot number holding the cached data
47481|     ULONG CacheNode;    // granule number in
    | cache file for this data granule
47482|     ULONG DataChecksum; // checksum of cached
    | data granule

```

```

47483|     ULONG Flags;           // Flags of granule in
    | cache file (normal, virtual write, etc...)
47484| } tDiskNode, *pDiskNode;
47485|
47486| typedef struct sIndexSectorPrefix {
47487|     ULONG      IndexChecksum; // checksum of
    | rest of structure (i.e. 512-4 = 508 bytes), 0=ignore
47488|     ULONG      PrefixSize; // how big this
    | structure is (so it can be skipped to find disk nodes)
47489|     ULONG      Version;      // version
    | number for index file format
47490|     ULONG      Sequence;     // counter
    | incremented every time we write an index sector
47491|     LARGE_INTEGER DateTimeWritten; // when
    | this index sector was written (or zero for
    | End-Of-Index-File marker)
47492| } tIndexSectorPrefix, *pIndexSectorPrefix;
47493|
47494| #define PSM_INDEX_VERSION_2      (2)
47495| #define PSM_HEADER_VERSION_2    (2)
47496| #define PSM_HEADER_CURRENT_VERSION
    | PSM_HEADER_VERSION_2
47497| #define PSM_INDEX_CURRENT_VERSION
    | PSM_INDEX_VERSION_2
47498| #define DISK_NODES_PER_INDEX_SECTOR
    | ((512-sizeof(tIndexSectorPrefix)) / sizeof(tDiskNode))
47499|
47500| typedef struct sIndexSectorInfo {
47501|     tIndexSectorPrefix Prefix;
47502|     tDiskNode          DiskNode
    | [DISK_NODES_PER_INDEX_SECTOR];
47503| } tIndexSectorInfo, *pIndexSectorInfo;
47504|
47505| typedef union uIndexSector {
47506|     tIndexSectorInfo Info;           // the
    | actual data we care about
47507|     unsigned char Filler [512];     // pads the
    | structure to be 512 byte sector size
47508| } tIndexSector, *pIndexSector;
47509|
47510| typedef struct sShared {
47511|     ULONG      Count;
47512|     ERESOURCE  TreeResource;
47513|     tTree      Tree;
47514|     PRTL_BITMAP Map;
47515|     PRTL_BITMAP MapInTransform;
47516|     ULONG      HighestSequence;
47517|     BOOLEAN     RecycleNeeded;
47518|     keyType     LastDirtyKey;
47519|     keyType     HighestKeyKnownClean;

```



```

47520|     tTree    VirtualWritesTree;
47521|     ULONG    ClusterSize;
47522| } tShared, *pShared;
47523|
47524|
47525| #endif /* __PSM_HEADER_DATA_STRUCTURES */
47526|
47527| /*--- end of file header.h ---*/
47528|
47529|
47530|
47531| File Listing: IOCTL.h
47532|
47533| #include <devioctl.h>
47534|
47535| /*lint -save -e43*/
47536| // psm ioctls
47537| #define IOCTL_GET_VERSION
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA0,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47538| #define IOCTL_TURNON_PSM
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA1,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47539| #define IOCTL_TURNOFF_PSM
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA2,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47540| #define IOCTL_GET_ERROR
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA3,METHOD_OUT_DIRECT,FIL
    | E_ANY_ACCESS)
47541| //#define IOCTL_GET_STATS
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA4,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47542| #define IOCTL_SET_WIN32_LINK
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA6,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47543| #define IOCTL_OPEN_EX
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA7,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47544| #define IOCTL_CLOSE_EX
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA8,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47545| #define IOCTL_FREE_VOLUME
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA9,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47546| #define IOCTL_FREE_RANGES
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAB,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47547| #define IOCTL_GET_PROGRESS
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAC,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)

```

```

47548| //define IOCTL_GET_PSM_INFO
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAD,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47549| #define IOCTL_SET_FLUSH_ROUTINE
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAE,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47550| #define IOCTL_GET_VOLUME_STATS
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAF,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47551| #define IOCTL_GET_VOLUME_INFO
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB0,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47552| #define IOCTL_GET_PERSISTENT_SNAPSHOTS
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB1,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47553| #define IOCTL_GET_KERNEL_SNAPSHOT_INFO
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB2,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47554| #define IOCTL_GET_KERNEL_SNAPSHOT_VOLUMES
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB3,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47555| #define IOCTL_GET_USER_NAME
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB4,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47556| #define IOCTL_SET_USER_NAME
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB5,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47557| // vdisk ioctls
47558| #define IOCTL_UNWRITE_PROTECT
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB6,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47559| #define IOCTL_WRITE_PROTECT
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB7,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47560| #define IOCTL_IS_PSM_VOLUME
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB8,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47561|
47562| // back to internal ioctl codes
47563| #define IOCTL_SET_KERNEL_SNAPSHOT_INFO
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC0,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47564| #define IOCTL_REVERT_TO_PRISTINE
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC1,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47565| #define IOCTL_REVERT_TO_SNAPSHOT
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC2,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47566| #define IOCTL_INFORM_SYSTEM_READY
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC3,METHOD_BUFFERED,FILE_

```

```

    | ANY_ACCESS)
47567| #define IOCTL_LOG_EVENT
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC4,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47568| #define IOCTL_GET_VOLUME_CACHE_INFO
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC5,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47569| #define IOCTL_GET_PSM_EVENT
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC6,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47570| #define IOCTL_CREATE_FILES
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC7,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47571|
47572| #ifdef _DEBUG
47573| #define IOCTL_TEST_FUNCTION
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xFFD,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47574| #define IOCTL_BUG_CHECK
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xFFE,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47575| #endif
47576| #if MEMDBG
47577| #define IOCTL_GET_MEMORY_USAGE
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xFFF,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
47578| #endif
47579|
47580| typedef ULONG (*PUSER_MODE_ROUTINE)( PVOID Context );
47581|
47582| #define
    | RunInRing0(Driver,Routine,Context,BytesReturned)
    | DeviceIoControl(hDriver,IOCTL_RUN_IN_RING0,Context,sizeo
    | f(PVOID),Routine,sizeof(PVOID),BytesReturned)
47583|
47584| #define PSM_IFLAG_NEW_SNAPSHOT    0x01
47585| #define PSM_IFLAG_PERSISTENT      0x02
47586| #define PSM_IFLAG_SAVE_TEMP_ON_EXIT 0x80
47587|
47588| #define LINK_SetLink 0
47589| #define LINK_DeleteLink 1
47590|
47591| #define PSM_EVENT_CLEAN_CLUSTER_REGISTRY 1
47592| #define PSM_EVENT_VOLUME_ONLINE          2
47593|
47594| #define LENGTH_OF_UNIQUE
    | (((16*sizeof(WCHAR))*2)+(sizeof(WCHAR)*2))
47595|
47596| typedef struct sPSM_GetPSMEvent {
47597|     ULONG Event;

```

```

47598|    ULONG KernelSnapShotPointer;
47599|    WCHAR UniqueId[LENGTH_OF_UNIQUE]; // string in the
    | form of '123456781234567812345678'
47600|    WCHAR VolumeGuid[200];
47601|    WCHAR NTName[200];
47602|
47603|    union {
47604|        struct {
47605|            ULONG Stuff;
47606|        } CleanClusterRegistry;
47607|        struct {
47608|            ULONG Stuff;
47609|        } VolumeOnline;
47610|        struct {
47611|            char DoNotUse;
47612|        } DoNotUse;
47613|    };
47614| } tPSM_GetPSMEvent,*pPSM_GetPSMEvent;
47615|
47616| typedef struct sPSM_SetWin32Link {
47617|     ULONG Operation;
47618|     WCHAR Win32Link[256];
47619|     WCHAR NTDeviceName[256];
47620| } tPSM_SetWin32Link,*pPSM_SetWin32Link;
47621|
47622| typedef struct sPSM_GetVolumeCacheInfoIn {
47623|     WCHAR VolumeName[256];
47624| } tPSM_GetVolumeCacheInfoIn,
    | *pPSM_GetVolumeCacheInfoIn;
47625|
47626| typedef struct sPSM_FreeVolume {
47627|     PVOID      KernelSnapShotPointer;
47628|     WCHAR      VolumeName[256];
47629| } tPSM_FreeVolume, *pPSM_FreeVolume;
47630|
47631| typedef struct sPSM_SnapShotPointer {
47632|     PVOID      KernelSnapShotPointer;
47633| } tPSM_SnapShotPointer, *pPSM_SnapShotPointer;
47634|
47635| typedef struct sPSM_RevertToSnapShotIn {
47636|     PVOID      KernelSnapShotPointer;
47637|     ULONG      RevertFlags;
47638| } tPSM_RevertToSnapShotIn, *pPSM_RevertToSnapShotIn;
47639|
47640| #define tPSM_GetKernelSnapShotInfoIn
    | tPSM_SnapShotPointer
47641| #define sPSM_GetKernelSnapShotInfoIn
    | sPSM_SnapShotPointer
47642| #define pPSM_GetKernelSnapShotInfoIn
    | pPSM_SnapShotPointer

```

```

47643|
47644| typedef struct sPSM_SetUserNameIn {
47645|     PVOID          KernelSnapShotPointer;
47646|     WCHAR          Name[256];
47647| } tPSM_SetUserNameIn, *pPSM_SetUserNameIn;
47648|
47649| typedef struct sPSM_SetKernelSnapShotInfoIn {
47650|     PVOID          KernelSnapShotPointer;
47651|     tPSM_SetKernelSnapShotInfo Info;
47652| } tPSM_SetKernelSnapShotInfoIn,
    | *pPSM_SetKernelSnapShotInfoIn;
47653|
47654| typedef struct sPSM_GetKernelSnapShotInfoOut {
47655|     LARGE_INTEGER  SnapShotTime; // time snapshot
    | occurred.
47656|     ULONG          Instance;      // instance number
    | for this snapshot for volume mapping
47657|     NTSTATUS       Status;        // status that
    | caused this snapshot to be canceled.
47658|     ULONG          Persistent;
47659|     PVOID          DllPrivateUse;
47660|     PVOID          CallerPrivateUse;
47661|     ULONG          NumToKeep;
47662|     BYTE           Priority;
47663|     BYTE           SnapShotFlags;
47664|     BYTE           Reserved1;
47665|     BYTE           Reserved2; // padding
47666|     WCHAR          UserSnapShotName[256];
47667| } tPSM_GetKernelSnapShotInfoOut,
    | *pPSM_GetKernelSnapShotInfoOut;
47668|
47669| #pragma warning (push)
47670| #pragma warning (disable:4200)
47671| /*lint -save -e43 -e1501 */
47672|
47673| typedef struct sOpenTransactionInInternal {
47674|     DWORD          Size;
47675|     WCHAR          CacheFileName[256];
47676|     DWORD          SizeOfCacheFileMB;
47677|     DWORD          MaxSizeOfCacheFileMB;
47678|     DWORD          Flags;
47679|     DWORD          InternalFlags;
47680|     DWORD          QuiescentWait;
47681|     DWORD          QuiescentTimeout;
47682|     HANDLE         ErrorEvent;
47683|     HANDLE         AbortEvent;
47684|     PVOID          DllPrivateUse;
47685|     PVOID          CallerPrivateUse;
47686|     ULONG          NumToKeep;
47687|     BYTE           Priority;

```

```

47688|  BYTE      SnapShotFlags;
47689|  BYTE      Reserved1;
47690|  BYTE      Reserved2; // padding
47691|  ULONG      NumberOfDevices;
47692|  ULONG      DeviceName[]; // offset from
    | beginning of this structure
47693| } tOpenTransactionInInternal,
    | *pOpenTransactionInInternal;
47694|
47695| typedef struct sOpenTransactionOutInternal {
47696|  PVOID KernelSnapShotPointer; // cant access kernel
    | mode addresses in user mode
47697|  LARGE_INTEGER SnapShotTime; // time snapshot
    | occurred.
47698|  ULONG      Instance;    // instance number
    | for this snapshot for volume mapping
47699|  WCHAR      CacheFileName[256]; // cache file
    | name being used
47700|  ULONG      Persistent;
47701|  ULONG      NumberOfDevices;
47702|  ULONG      DeviceName[]; // offset from
    | beginning of this structure
47703| } tOpenTransactionOutInternal,
    | *pOpenTransactionOutInternal;
47704|
47705| typedef struct sPSM_LogEvent {
47706|  ULONG EventId;
47707|  ULONG Status;
47708|  ULONG NumStrings;
47709|  WCHAR *Strings[];
47710| } tPSM_LogEvent, *pPSM_LogEvent;
47711|
47712| /*lint -restore*/
47713| #pragma warning (pop)
47714|
47715| typedef struct sClosePSMInternal {
47716|  PVOID KernelSnapShotPointer; // cant access kernel
    | mode addresses in user mode
47717| } tClosePSMInternal, *pClosePSMInternal;
47718|
47719| typedef struct sPSMVolumeInfoln {
47720|  unsigned short Size;
47721|  unsigned short Version;
47722|  WCHAR VolumeName[256];
47723| } tPSMVolumeInfoln, *pPSMVolumeInfoln;
47724|
47725| typedef struct sPSM_GetPersistentSnapShotsOut {
47726|  PVOID KernelPointers[MAX_NUMBER_OF_SNAPSHOTS];
47727| } tPSM_GetPersistentSnapShotsOut,
    | *pPSM_GetPersistentSnapShotsOut;

```

```

47728|
47729| typedef struct sPSMVolumeInfoOut {
47730|     ULONG SectorsPerCluster;
47731|     ULONG Cluster0Offset;
47732| } tPSMVolumeInfoOut, *pPSMVolumeInfoOut;
47733|
47734| #if MEMDBG
47735| typedef struct sGetMemoryUsageOut {
47736|     ULONG MemTotalNonpagedAlloced;
47737|     ULONG MemTotalPagedAlloced;
47738|     ULONG MemMaxNonpagedAlloced;
47739|     ULONG MemMaxPagedAlloced;
47740| } tGetMemoryUsageOut, *pGetMemoryUsageOut;
47741|
47742| #endif
47743|
47744| typedef struct sSetFlushRoutine {
47745|     NTSTATUS (*ZwFlushBuffersFile)( IN HANDLE
        | FileHandle, OUT PIO_STATUS_BLOCK IoStatusBlock );
47746| } tSetFlushRoutine, *pSetFlushRoutine;
47747|
47748| /*lint -restore*/
47749|
47750|
47751|
47752| File Listing: iodirect.cpp
47753|
47754| #include "precomp.h"
47755|
47756|
47757| ULONG PSMDirectIOOptions=0;
47758|
47759| //-----
        | -----
47760|
47761| void MapDirectIo::reset()
47762| {
47763|     Profile("MapDirectIo::reset");
47764|     //Debug(DEBUG_DICT,("MapDirectIo::reset\n"));
47765|     #if 0
47766|         rbtree_Init(&tree);
47767|     #else
47768|         __try {
47769|             tTreeLeaf *node = tree.HeadLeaf;
47770|             while ( node != tree.TailLeaf ) {
47771|                 keyType searchKey = node->Key;
47772|                 tTreeLeaf *delnode = rbtree_Delete(&tree,
        | searchKey);
47773|                 ASSERT ( delnode != NULL );
47774|                 if ( delnode != NULL ) {

```

```

47775|         ASSERT ( delnode->Key == searchKey );
47776|         FreeNode (delnode);
47777|     }
47778|     node = tree.HeadLeaf;
47779| }
47780|     rbtree_Init(&tree);
47781| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
47782|     Debug(DEBUG_DICT,("Exception %08x in
    | MapDirectIo::reset\n",GetExceptionCode()));
47783| }
47784| #endif
47785| }
47786|
47787| //-----
    | -----
47788|
47789| MapFileToDisk::MapFileToDisk():
47790|     openFlag (false),
47791|     inverseMapFlag (false),
47792|     clusterSizeInBytes (0),
47793|     sectorSizeInBytes (0)
47794| {
47795|     Profile("MapFileToDisk::MapFileToDisk");
47796| //
    | Debug(DEBUG_DICT,("MapFileToDisk::MapFileToDisk()\n"));
47797|     rbtree_Init (&tree);
47798| }
47799|
47800| //-----
    | -----
47801|
47802| MapFileToDisk::~MapFileToDisk()
47803| {
47804|     Profile("MapFileToDisk::~MapFileToDisk");
47805| //
    | Debug(DEBUG_DICT,("MapFileToDisk::~MapFileToDisk\n"));
47806|     close();
47807| }
47808|
47809| //-----
    | -----
47810|
47811| void MapFileToDisk::close()
47812| {
47813|     Profile("MapFileToDisk::close");
47814| //     Debug(DEBUG_DICT,("MapFileToDisk::close
    | (open=%d)\n",openFlag));
47815|     if ( openFlag ) {
47816|         reset();

```



```

47817|     openFlag = false;
47818| }
47819| }
47820|
47821| //-----
| -----
47822|
47823| NTSTATUS MapFileToDisk::SetInternal (
47824|     PDEVICE_OBJECT      DeviceObject,
47825|     PFILE_OBJECT         FileObject,
47826|     RETRIEVAL_POINTERS_BUFFER *rp,
47827|     ULONG                 ClusterSizeInBytes,
47828|     ULONG                 SectorSizeInBytes,
47829|     bool                  ReverseMapping )
47830| {
47831|     NTSTATUS status = STATUS_SUCCESS;
47832|
47833|     Profile("MapFileToDisk::SetInternal");
47834|     Debug(DEBUG_DICT,("MapFileToDisk::SetInternal\n"));
47835|     ASSERT ( !openFlag );
47836|     ASSERT ( ReverseMapping == inverseMapFlag );
47837|
47838|     if ( openFlag ) {
47839|         status = STATUS_UNSUCCESSFUL;
47840|         Debug(DEBUG_DEVCON,("Attempt to call
| MapFileToDisk::SetInternal on already open
| object!\n"));
47841|     } else {
47842|         reset();
47843|         LARGE_INTEGER CurrentVcn = rp->StartingVcn;
47844|         LARGE_INTEGER NumClusters = {0};
47845|         for ( ULONG i=0; i < rp->ExtentCount; ++i ) {
47846|             NumClusters.QuadPart =
| rp->Extents[i].NextVcn.QuadPart - CurrentVcn.QuadPart;
47847|             if ( ReverseMapping ) {
47848|                 // Weirdness alert: when ReverseMapping
| is true, we swap disk and file clusters
47849|                 // in the mapping. This way we can ask
| for a file offset given a disk offset.
47850|                 status = insertExtent (
| rp->Extents[i].Lcn, CurrentVcn, NumClusters );
47851|             } else {
47852|                 status = insertExtent ( CurrentVcn,
| rp->Extents[i].Lcn, NumClusters );
47853|             }
47854|             if ( status != STATUS_SUCCESS ) {
47855|                 break;
47856|             }
47857|             CurrentVcn = rp->Extents[i].NextVcn;
47858|         }

```

```

47859|
47860|     if ( status == STATUS_SUCCESS ) {
47861|         openFlag = true;
47862|         deviceObject    = DeviceObject;
47863|         fileObject      = FileObject;
47864|         clusterSizeInBytes = ClusterSizeInBytes;
47865|         sectorSizeInBytes = SectorSizeInBytes;
47866|     }
47867| }
47868|
47869| return status;
47870| }
47871|
47872| //-----
| -----
47873|
47874| NTSTATUS MapFileToDisk::insertExtent (
47875|     LARGE_INTEGER FileClusterOffset,
47876|     LARGE_INTEGER DiskClusterOffset,
47877|     LARGE_INTEGER NumClusters )
47878| {
47879|     NTSTATUS Status = STATUS_SUCCESS;
47880|     Profile("MapFileToDisk::insertExtent");
47881|
47882| #ifdef DEBUG
47883|     Debug(DEBUG_DEVCON,("mftd::insertExtent:
| FileCluster=%l64x, DiskCluster=%l64x,
| NumClusters=%l64x\n",
47884|         FileClusterOffset.QuadPart,
47885|         DiskClusterOffset.QuadPart,
47886|         NumClusters.QuadPart));
47887| #endif /*DEBUG*/
47888|
47889|     LARGE_INTEGER ClustersLeft = NumClusters;
47890|     while ( ClustersLeft.QuadPart > 0 ) {
47891|         pTreeLeaf node = AllocNode();
47892|         if ( node ) {
47893|             LARGE_INTEGER ClustersEncoded =
| ClustersLeft;
47894|             if ( ClustersEncoded.QuadPart >=
| IODIRECT_MAX_CLUSTER_RUN ) {
47895|                 ClustersEncoded.QuadPart =
| IODIRECT_MAX_CLUSTER_RUN - 1;
47896|             }
47897|
47898|             // Now the weird part: rearranging the
| bits we need into the rbtree node.
47899|             // Because we will later use
| rbtree_SearchUpperBound, we have to encode the
47900|             // last cluster in the extent, not the

```

```

| first.
47901|         node->Key = (FileClusterOffset.QuadPart +
| ClustersEncoded.QuadPart - 1) <<
| (IODIRECT_BITS_PER_CLUSTER_RUN + 8);
47902|         node->Key |= ClustersEncoded.QuadPart << 8;
47903|         node->Key |= (DiskClusterOffset.QuadPart >>
| 31) & 0xff;
47904|
47905|         ASSERT ( ((DiskClusterOffset.QuadPart >>
| 31) & ~__int64(0xff)) == 0 );
47906|
47907|         node->Pos = DiskClusterOffset.LowPart &
| 0x7fffffff;
47908|
47909| #ifdef DEBUG
47910|         Debug(DEBUG_DEVCON,("mftd::insertExtent:
| node->Key=%l64x, node->Pos=%x\n",node->Key,node->Pos));
47911|         LARGE_INTEGER VerifyClustersEncoded,
| VerifyFileCluster, VerifyDiskCluster;
47912|         // Extract the information back out of the
| node the same way we will later when performing I/O.
47913|         // Make sure we can get back exactly what
| we put in.
47914|         VerifyClustersEncoded.QuadPart = (node->Key
| >> 8) & (IODIRECT_MAX_CLUSTER_RUN - 1);
47915|         VerifyFileCluster.QuadPart = node->Key >>
| (IODIRECT_BITS_PER_CLUSTER_RUN + 8);
47916|         VerifyFileCluster.QuadPart -=
| (VerifyClustersEncoded.QuadPart - 1);
47917|         VerifyDiskCluster.QuadPart = ((node->Key &
| 0xff) << 31) | __int64(node->Pos);
47918|         ASSERT ( VerifyClustersEncoded.QuadPart ==
| ClustersEncoded.QuadPart );
47919|         ASSERT ( VerifyFileCluster.QuadPart ==
| FileClusterOffset.QuadPart );
47920|         ASSERT ( VerifyDiskCluster.QuadPart ==
| DiskClusterOffset.QuadPart );
47921| #endif /*DEBUG*/
47922|         int result = rbtree_Insert ( &tree, node );
47923|         ASSERT(result==0);
47924|         if ( result != 0 ) {
47925|             Debug(DEBUG_DEVCON,("!!!
| MapFileToDisk::insertExtent: Failure to insert rbtree
| node\n"));
47926|             Debug(DEBUG_DEVCON,("!!! Key=%016l64x,
| Pos=%08x\n", node->Key, node->Pos));
47927|             FreeNode(node);
47928|             node = 0;
47929|             Status = STATUS_UNSUCCESSFUL;
47930|             break;

```

```

47931|         }
47932|
47933|         ClustersLeft.QuadPart -=
| ClustersEncoded.QuadPart;
47934|         FileClusterOffset.QuadPart +=
| ClustersEncoded.QuadPart;
47935|         DiskClusterOffset.QuadPart +=
| ClustersEncoded.QuadPart;
47936|     } else {
47937|         Status = STATUS_INSUFFICIENT_RESOURCES;
47938|         Debug(DEBUG_DEVCON,("!!!
| MapFileToDisk::insertExtent: Could not allocate rbtree
| node!\n"));
47939|         break;
47940|     }
47941| }
47942|
47943| return Status;
47944| }
47945|
47946| //-----
| -----
47947|
47948| NTSTATUS MapFileToDisk::getDiskOffsetForFileOffset (
47949|     LARGE_INTEGER FileOffsetInBytes,
47950|     LARGE_INTEGER &DiskOffsetInBytes,
47951|     LARGE_INTEGER &ExtentLengthInBytes,
47952|     bool          SuppressDebugErrors )
47953| {
47954|     NTSTATUS status = STATUS_NOT_FOUND;
47955|     DiskOffsetInBytes.QuadPart = __int64(-1);
47956|     ExtentLengthInBytes.QuadPart = 0;
47957|
47958|     | Profile("MapFileToDisk::getDiskOffsetForFileOffset");
47959|     ASSERT ( openFlag );
47960|     ASSERT ( clusterSizeInBytes > 0 );
47961|     ASSERT ( sectorSizeInBytes > 0 );
47962|     ASSERT ( (sectorSizeInBytes &
| (sectorSizeInBytes-1)) == 0 ); // make sure it's a
| power of 2
47963|     ASSERT ( clusterSizeInBytes % sectorSizeInBytes ==
| 0 );
47964|     ASSERT ( clusterSizeInBytes >= sectorSizeInBytes );
47965|     ASSERT ( FileOffsetInBytes.QuadPart %
| sectorSizeInBytes == 0 );
47966|
47967|     LARGE_INTEGER FileCluster;
47968|     FileCluster.QuadPart = FileOffsetInBytes.QuadPart /
| clusterSizeInBytes;

```

```

47969|    ULONG OffsetIntoCluster = ULONG
      | (FileOffsetInBytes.QuadPart % clusterSizeInBytes);
47970|
47971|    tTreeLeaf *node = rbtree_SearchUpperBound (
47972|        &tree,
47973|        FileCluster.QuadPart <<
      | (IODIRECT_BITS_PER_CLUSTER_RUN + 8) );
47974|
47975|    if ( node ) {
47976|        LARGE_INTEGER FileClusterFound;
47977|        FileClusterFound.QuadPart = node->Key >>
      | (IODIRECT_BITS_PER_CLUSTER_RUN + 8);
47978|        ASSERT ( FileClusterFound.QuadPart >=
      | FileCluster.QuadPart );
47979|
47980|        LARGE_INTEGER NumClusters;
47981|        NumClusters.QuadPart = (node->Key >> 8) &
      | (IODIRECT_MAX_CLUSTER_RUN - 1);
47982|        ASSERT ( NumClusters.QuadPart > 0 );
47983|
47984|        LARGE_INTEGER DiskCluster;
47985|        DiskCluster.QuadPart = ((node->Key & 0xff) <<
      | 31) | __int64(node->Pos);
47986|
47987|        // The file cluster we stored in the node is
      | actually the last cluster in the extent.
47988|        // Convert it to the first cluster.
47989|        FileClusterFound.QuadPart -=
      | (NumClusters.QuadPart - 1);
47990|
47991|        // The requested file offset had better be
      | within the extent we found...
47992|        // This means the extent we found must be at or
      | before the requested cluster,
47993|        // *and* it must end at or after the requested
      | cluster.
47994|        ULONG ExtentStartOk = FileClusterFound.QuadPart
      | <= FileCluster.QuadPart;
47995|        ULONG ExtentEndOk = FileClusterFound.QuadPart
      | + NumClusters.QuadPart > FileCluster.QuadPart;
47996|        if ( ExtentStartOk && ExtentEndOk ) {
47997|            status = STATUS_SUCCESS; // We ROCK!!!
47998|
47999|            __int64 OffsetIntoExtent =
48000|                (FileCluster.QuadPart -
      | FileClusterFound.QuadPart)*clusterSizeInBytes +
48001|                OffsetIntoCluster;
48002|
48003|            ExtentLengthInBytes.QuadPart =
      | NumClusters.QuadPart * clusterSizeInBytes;

```

```

48004|         ExtentLengthInBytes.QuadPart -=
        | OffsetIntoExtent;
48005|
48006|         DiskOffsetInBytes.QuadPart =
        | DiskCluster.QuadPart * clusterSizeInBytes;
48007|         DiskOffsetInBytes.QuadPart +=
        | OffsetIntoExtent;
48008|
48009|         if ( DiskOffsetInBytes.QuadPart < 0 ) {
48010|             Debug(DEBUG_DEVCON,("!!!
        | mftd::getDiskOffsetForFileOffset: Negative
        | DiskOffsetInBytes %016l64x (dec
        | %l64d)\n",DiskOffsetInBytes.QuadPart,DiskOffsetInBytes.Q
        | uadPart));
48011|             Debug(DEBUG_DEVCON,("!!!
        | node->Key=%016l64x, node->Pos=%08x,
        | FileOffsetInBytes=%016l64x, ClusterSize=%08x\n",
48012|                 node->Key,
48013|                 node->Pos,
48014|                 FileOffsetInBytes.QuadPart,
48015|                 clusterSizeInBytes));
48016|             status = PSM_ERROR_UNSUCCESSFUL;
48017|         }
48018|
48019|         ASSERT ( ExtentLengthInBytes.QuadPart > 0
        | );
48020|         ASSERT ( DiskOffsetInBytes.QuadPart >= 0 );
48021|         ASSERT ( ExtentLengthInBytes.QuadPart %
        | sectorSizeInBytes == 0 );
48022|         ASSERT ( DiskOffsetInBytes.QuadPart %
        | sectorSizeInBytes == 0 );
48023|     } else {
48024|         if ( !inverseMapFlag ) {
48025|             // Only a problem if we are in forward
        | map mode: i.e. trying to read/write to file
48026|             Debug(DEBUG_DEVCON,("!!!
        | MapFileToDisk::getDiskOffsetForFileOffset: file cluster
        | %016l64x is outside extent (start=%016l64x,
        | count=%016l64x)\n",
48027|                 FileCluster.QuadPart,
48028|                 FileClusterFound.QuadPart,
48029|                 NumClusters.QuadPart));
48030|             Debug(DEBUG_DEVCON,("!!!
        | ExtentStartOk=%S, ExtentEndOk=%S\n",
48031|                 (ExtentStartOk?"TRUE":"FALSE"),
48032|                 (ExtentEndOk?"TRUE":"FALSE")));
48033|             ASSERT (FALSE);
48034|         }
48035|     }
48036| } else {

```

```

48037|         if ( !inverseMapFlag && !SuppressDebugErrors )
48038|         | {
48038|             // Only a problem if we are in forward map
48039|             | mode: i.e. trying to read/write to file.
48039|             Debug(DEBUG_DEVCON,("!!!
48040|             | MapFileToDisk::getDiskOffsetForFileOffset: Could not
48041|             | find file offset %016l64x in
48042|             | rbtree\n",FileOffsetInBytes.QuadPart));
48043|         }
48044|     }
48045|
48046|     return status;
48047| }
48048|
48049| NTSTATUS DirectAccessFile::SetInternal (
48050|     PDEVICE_OBJECT      Volume,
48051|     PFILE_OBJECT         FileObject,
48052|     RETRIEVAL_POINTERS_BUFFER *rp,
48053|     ULONG                 ClusterSizeInBytes,
48054|     ULONG                 SectorSizeInBytes )
48055| {
48056|     NTSTATUS Status = STATUS_SUCCESS;
48057|     Profile("DirectAccessFile::SetInternal");
48058|     | Debug(DEBUG_DICT,("DirectAccessFile::SetInternal\n"));
48059|     ASSERT ( !isOpen );
48060|
48061|     Status = fileMap.SetInternal (
48062|         Volume,
48063|         FileObject,
48064|         rp,
48065|         ClusterSizeInBytes,
48066|         SectorSizeInBytes,
48067|         false );
48068|
48069|     if ( Status == STATUS_SUCCESS ) {
48070|         Status = diskMap.SetInternal (
48071|             NULL,
48072|             NULL,
48073|             rp,
48074|             ClusterSizeInBytes,
48075|             SectorSizeInBytes,
48076|             true );
48077|
48078|         if ( Status == STATUS_SUCCESS ) {
48079|             isOpen = true;

```

```

48080|     }
48081| }
48082|
48083| return Status;
48084| }
48085|
48086| //-----
| -----
48087|
48088| void DirectAccessFile::close()
48089| {
48090|     Profile("DirectAccessFile::close");
48091|     Debug(DEBUG_DICT,("DirectAccessFile::close\n"));
48092|     if(isOpen) {
48093|         fileMap.close();
48094|         diskMap.close();
48095|         isOpen = false;
48096|     }
48097| }
48098|
48099| //-----
| -----
48100|
48101| NTSTATUS DirectAccessFile::perform_io (
48102|     PVOID          Buffer,
48103|     LARGE_INTEGER  FileOffsetInBytes,
48104|     ULONG          BytesToProcess,
48105|     DirectAccessFile::IOTYPE IoType,
48106|     const char      *IoTypeString )
48107| {
48108|     Profile("DirectAccessFile::perform_io");
48109|     ASSERT ( IoType==IOTYPE_READ ||
| IoType==IOTYPE_WRITE );
48110|     ASSERT ( IoTypeString != NULL );
48111|     if ( IoType!=IOTYPE_READ && IoType!=IOTYPE_WRITE )
| {
48112|         Debug(DEBUG_DEVCON,("!!!
| DirectAccessFile::perform_io: Invalid
| IoType=%08x\n",IoType));
48113|         return STATUS_INVALID_PARAMETER;
48114|     }
48115|
48116|     NTSTATUS Status = STATUS_SUCCESS;
48117|     if ( isOpen ) {
48118|         const ULONG SectorSize =
| fileMap.getSectorSizeInBytes();
48119|         ASSERT ( SectorSize > 0 );
48120|         if ( BytesToProcess % SectorSize != 0 ) {
48121|             Status = STATUS_INVALID_PARAMETER;
48122|             Debug(DEBUG_DEVCON,("DirectAccessFile::%s:

```



```

    | Invalid
    | BytesToProcess=%08x\n",IoTypeString,BytesToProcess));
48123|     } else if ( FileOffsetInBytes.QuadPart %
    | SectorSize != 0 ) {
48124|         Status = STATUS_INVALID_PARAMETER;
48125|         Debug(DEBUG_DEVCON,("DirectAccessFile::%s:
    | Invalid
    | FileOffset=%016l64x\n",IoTypeString,FileOffsetInBytes.Qu
    | adPart));
48126|     } else {
48127|         LARGE_INTEGER DiskOffsetInBytes = {0};
48128|         LARGE_INTEGER ExtentLengthInBytes = {0};
48129|         ULONG BytesLeftToProcess = BytesToProcess;
48130|         char *BufferPointer = (char *) Buffer;
48131|
48132|         while ( BytesLeftToProcess > 0 ) {
48133|             Status =
    | fileMap.getDiskOffsetForFileOffset (
48134|                 FileOffsetInBytes,
48135|                 DiskOffsetInBytes,
48136|                 ExtentLengthInBytes,
48137|                 false );
48138|
48139|             if ( Status == STATUS_SUCCESS ) {
48140|                 ULONG BytesToProcess =
    | BytesLeftToProcess;
48141|                 if ( BytesToProcess >
    | ExtentLengthInBytes.QuadPart ) {
48142|                     BytesToProcess =
    | ULONG(ExtentLengthInBytes.QuadPart);
48143|                 }
48144|
48145|                 switch ( IoType ) {
48146|                     case IOTYPE_READ: {
48147|                         Status = Sblo_ReadDevice(
48148|
    | ((PFILTERED_EXTENSION)(GetDeviceExtension(fileMap.GetDev
    | iceObject())))->TargetDeviceObject,
48149|                             &DiskOffsetInBytes,
48150|                             BytesToProcess,
48151|                             BufferPointer );
48152|                     } break;
48153|
48154|                     case IOTYPE_WRITE: {
48155|                         Status = Sblo_WriteDevice(
48156|
    | ((PFILTERED_EXTENSION)(GetDeviceExtension(fileMap.GetDev
    | iceObject())))->TargetDeviceObject,
48157|                             &DiskOffsetInBytes,
48158|                             BytesToProcess,

```

```

48159|                (const char
    | *)BufferPointer);
48160|                } break;
48161|
48162|                default: { // should not be
    | possible to get here because we already check for this
48163|                ASSERT(FALSE);
48164|                Status =
    | STATUS_INVALID_PARAMETER;
48165|                }
48166|            }
48167|
48168|            if(!INT_SUCCESS(Status)) {
48169|
    | Debug(DEBUG_DEVCON,("DirectAccessFile::%s: Error %08x
    | in disk I/O\n",IoTypeString,Status));
48170|                break;
48171|            }
48172|
48173|            BufferPointer += BytesToProcess;
48174|            FileOffsetInBytes.QuadPart +=
    | BytesToProcess;
48175|            BytesLeftToProcess -=
    | BytesToProcess;
48176|        } else {
48177|
    | Debug(DEBUG_DEVCON,("DirectAccessFile::%s: fileMap
    | returned %08x\n",IoTypeString,Status));
48178|            break;
48179|        }
48180|    } // while
48181|    }
48182| } else {
48183|     Status = STATUS_UNSUCCESSFUL;
48184|     Debug(DEBUG_DEVCON,("DirectAccessFile::%s: I'm
    | not open!\n",IoTypeString));
48185| }
48186|
48187| if ( !INT_SUCCESS(Status) ) {
48188|     Debug(DEBUG_DEVCON,("DirectAccessFile::%s:
    | returning %08x\n",IoTypeString,Status));
48189| }
48190|
48191| return Status;
48192| }
48193|
48194| //-----
    | -----
48195|
48196| NTSTATUS DirectAccessFile::read (

```

```

48197|  PVOID      Buffer,
48198|  LARGE_INTEGER FileOffsetInBytes,
48199|  ULONG      BytesToRead )
48200| {
48201|  Profile("DirectAccessFile::read");
48202|  return perform_io ( Buffer, FileOffsetInBytes,
    | BytesToRead, IOTYPE_READ, "read" );
48203| }
48204|
48205| //-----
    | -----
48206|
48207| NTSTATUS DirectAccessFile::write (
48208|  PCVOID      Buffer,
48209|  LARGE_INTEGER FileOffsetInBytes,
48210|  ULONG      BytesToWrite )
48211| {
48212|  Profile("DirectAccessFile::write");
48213|  return perform_io ( (PVOID)Buffer,
    | FileOffsetInBytes, BytesToWrite, IOTYPE_WRITE, "write"
    | );
48214| }
48215|
48216| //-----
    | -----
48217|
48218| NTSTATUS DirectAccessFile::getFileOffset (
48219|  LARGE_INTEGER DiskOffsetInBytes,
48220|  LARGE_INTEGER &FileOffsetInBytes,
48221|  LARGE_INTEGER &ExtentLengthInBytes )
48222| {
48223|  // Weirdness alert: we are using the diskMap
    | object in 'reverse mapping mode'.
48224|  // That means that we are passing in a disk offset,
    | though diskMap thinks it
48225|  // is a file offset, and vice versa.
48226|
48227|  Profile("DirectAccessFile::getFileOffset");
48228|
48229|  NTSTATUS status =
    | diskMap.getDiskOffsetForFileOffset (
48230|    DiskOffsetInBytes,
48231|    FileOffsetInBytes,
48232|    ExtentLengthInBytes,
48233|    false );
48234|
48235|  if ( NT_SUCCESS(status) ) {
48236|    Debug(DEBUG_DEVCON,("diskMap.getFileOffset
    | returned success - in diskofs=%016l64x, out
    | fileofs=%016l64x,

```



```

| <= 0 ) {
48274|
| Debug(DEBUG_DEVCON,("DirectAccessFile::getFileSizeInByte
| s: Invalid non-positive extent length %016l64x\n",
| ExtentLengthInBytes.QuadPart));
48275|         break;
48276|     }
48277|     } else if ( LookupStatus ==
| STATUS_NOT_FOUND ) {
48278|         // We are exactly at EOF...
48279|         SizeInBytes.QuadPart =
| FileOffsetInBytes.QuadPart;
48280|         status = STATUS_SUCCESS;
48281|         break;
48282|     } else {
48283|         // Something weird happened...
48284|         status = LookupStatus;
48285|
| Debug(DEBUG_DEVCON,("DirectAccessFile::getFileSizeInByte
| s: Weird LookupStatus=%08x\n",LookupStatus));
48286|         break;
48287|     }
48288| }
48289| } else {
48290|
| Debug(DEBUG_DEVCON,("DirectAccessFile::getFileSizeInByte
| s: this=%08x is not open!\n",this));
48291| }
48292|
48293|     ASSERT(NT_SUCCESS(status));
48294| } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
48295|     status = GetExceptionCode();
48296|     SizeInBytes.QuadPart = 0;
48297|     Debug(DEBUG_DEVCON,("!!! Exception %08x in
| DirectAccessFile::getFileSizeInBytes\n",status));
48298| }
48299|
48300|
| Debug(DEBUG_DEVCON,("DirectAccessFile::getFileSizeInByte
| s: status=%08x,
| Size=%016l64x\n",status,SizeInBytes.QuadPart));
48301|     return status;
48302| }
48303|
48304| //-----
| -----
48305|
48306| static const int    BITS_IN_KEY    = 64;
48307| static const int    BITS_IN_POS    = 31;

```

```

48308| static const int  BITS_PER_CLUSTER  = 39;
48309| static const int  BITS_PER_ID        = (BITS_IN_KEY
    | + BITS_IN_POS) - 2*BITS_PER_CLUSTER;
48310| static const CLUSTER_INDEX MAX_IDENTIFIER =
    | (CLUSTER_INDEX(1) << BITS_PER_ID) - 1;
48311| static const CLUSTER_INDEX MAX_VIRGIN_CLUSTERS =
    | CLUSTER_INDEX(1) << BITS_PER_CLUSTER;
48312|
48313| //-----
    | -----
48314|
48315| bool tVirginMap::isEmpty() const
48316| {
48317|     return tree.HeadLeaf == tree.TailLeaf;
48318| }
48319|
48320| //-----
    | -----
48321|
48322| NTSTATUS tVirginMap::validateExtent (
48323|     CLUSTER_INDEX  firstCluster,
48324|     CLUSTER_INDEX  numClusters ) const
48325| {
48326|     NTSTATUS status = STATUS_SUCCESS;
48327|     Profile("tVirginMap::validateExtent");
48328|
48329|     if ( numClusters<1 ||
        | numClusters>MAX_VIRGIN_CLUSTERS ) {
48330|         status = STATUS_UNSUCCESSFUL;
48331|         ASSERT(numClusters>=1);
48332|         ASSERT(numClusters<=MAX_VIRGIN_CLUSTERS);
48333|     } else if ( firstCluster<0 ||
        | firstCluster>=MAX_VIRGIN_CLUSTERS ) {
48334|         status = STATUS_UNSUCCESSFUL;
48335|         ASSERT(firstCluster>=0);
48336|         ASSERT(firstCluster<MAX_VIRGIN_CLUSTERS);
48337|     } else if ( totalClustersOnVolume != 0 ) {
48338|         // This means that the tVirginMap object knows
        | how big the volume's filesystem
48339|         // is, because it has been told by an outside
        | caller. This also means the
48340|         // client code wants this object to enforce
        | clusters to be within the valid range.
48341|         if ( firstCluster >= totalClustersOnVolume ) {
48342|             status = STATUS_INVALID_PARAMETER;
48343|             ASSERT(firstCluster <
        | totalClustersOnVolume);
48344|         } else if ( firstCluster+numClusters-1 >=
        | totalClustersOnVolume ) {
48345|             status = STATUS_INVALID_PARAMETER;

```

```

48346|         ASSERT(firstCluster+numClusters-1 <
| totalClustersOnVolume);
48347|     }
48348| }
48349|
48350| return status;
48351| }
48352|
48353| //-----
| -----
48354|
48355| NTSTATUS tVirginMap::insertExtent (
48356|     CLUSTER_INDEX firstCluster,
48357|     CLUSTER_INDEX numClusters )
48358| {
48359|     NTSTATUS status = STATUS_SUCCESS;
48360|     Profile("tVirginMap::insertExtent");
48361|     tTreeLeaf *node = AllocNode();
48362|     if ( node ) {
48363|         bool NodeWasInserted = false;
48364|         status = validateExtent (firstCluster,
| numClusters);
48365|         if ( NT_SUCCESS(status) ) {
48366|             ULONG identifier = 0;
48367|             status = findNextIdentifierForExtentLength
| (numClusters, firstCluster, identifier);
48368|             if ( NT_SUCCESS(status) ) {
48369|                 ULONG pos = 0;
48370|                 status = encodeNumbersIntoNode
| (node->Key, pos, firstCluster, numClusters,
| identifier);
48371|                 if ( NT_SUCCESS(status) ) {
48372|                     node->Pos = pos;
48373|                     int insertResult =
| rbtree_Insert(&tree,node);
48374|                     if ( insertResult == 0 ) {
48375|                         NodeWasInserted = true;
48376|                     } else {
48377|                         // This can happen when we run
| out of identifiers for a given number of clusters.
48378|                         // It's unfortunate, but not
| that big a deal because if we have 128K nodes with the
48379|                         // same extent length, chances
| are that extent length is not very big anyway, so we
48380|                         // are missing out on a small
| amount of virgin space.
48381|                         status = PSM_TREE_INSERT_ERROR;
48382|
| Debug(DEBUG_DEVCON,("tVirginMap::insertExtent:
| Insertion failure %08x for firstCluster=%016l64x,

```

```

    | numClusters=%016l64x\n",insertResult,firstCluster,numClu
    | sters));
48383|         }
48384|     }
48385| }
48386| }
48387|
48388|     if ( !NodeWasInserted ) {
48389|         FreeNode(node);
48390|     }
48391| } else {
48392|     status = STATUS_INSUFFICIENT_RESOURCES;
48393|     ASSERT(FALSE);
48394| }
48395| return status;
48396| }
48397|
48398| //-----
    | -----
48399|
48400| NTSTATUS tVirginMap::removeAnyExtent (
48401|     CLUSTER_INDEX  &firstCluster,
48402|     CLUSTER_INDEX  &numClusters )
48403| {
48404|     NTSTATUS status = STATUS_SUCCESS;
48405|     Profile("tVirginMap::removeAnyExtent");
48406|     firstCluster = numClusters = 0;
48407|     tTreeLeaf *node = tree.HeadLeaf;
48408|     if ( node != tree.TailLeaf ) {
48409|         status = removeExtentByKey (node->Key,
    | firstCluster, numClusters);
48410|     } else {
48411|         status = STATUS_UNSUCCESSFUL; // cannot
    | remove because tree is empty!
48412|     }
48413|
48414|     return status;
48415| }
48416|
48417| //-----
    | -----
48418|
48419| NTSTATUS tVirginMap::queryLongestExtent ( CLUSTER_INDEX
    | &numClusters ) const
48420| {
48421|     NTSTATUS status = STATUS_NOT_FOUND;
48422|     numClusters = 0;
48423|
48424|     Profile("tVirginMap::queryLongestExtent");
48425|     tTreeLeaf *node = tree.HeadLeaf;

```



```

48426|  if ( node != tree.TailLeaf ) {
48427|      while ( node->Right != tree.TailLeaf ) {
48428|          node = node->Right;
48429|      }
48430|
48431|      CLUSTER_INDEX firstCluster=0;
48432|      ULONG identifier=0;
48433|      status = decodeNumbersFromNode (node->Key,
    | node->Pos, firstCluster, numClusters, identifier);
48434|  }
48435|
48436|  return status;
48437| }
48438|
48439| //-----
    | -----
48440|
48441| NTSTATUS tVirginMap::removeLongestExtent (
48442|  CLUSTER_INDEX  &firstCluster,
48443|  CLUSTER_INDEX  &numClusters )
48444| {
48445|  NTSTATUS status = STATUS_SUCCESS;
48446|  Profile("tVirginMap::removeLongestExtent");
48447|
48448|  firstCluster = numClusters = 0;
48449|  tTreeLeaf *node = tree.HeadLeaf;
48450|  if ( node != tree.TailLeaf ) {
48451|      while ( node->Right != tree.TailLeaf ) {
48452|          node = node->Right;
48453|      }
48454|
48455|      status = removeExtentByKey (node->Key,
    | firstCluster, numClusters);
48456|  } else {
48457|      status = STATUS_UNSUCCESSFUL; // cannot
    | remove because tree is empty!
48458|  }
48459|
48460|  return status;
48461| }
48462|
48463| //-----
    | -----
48464|
48465| NTSTATUS tVirginMap::removeExtentByKey (
48466|  keyType      keyToRemove,
48467|  CLUSTER_INDEX  &firstCluster,
48468|  CLUSTER_INDEX  &numClusters )
48469| {
48470|  NTSTATUS status = STATUS_SUCCESS;

```

```

48471| Profile("tVirginMap::removeExtentByKey");
48472|
48473| tTreeLeaf *node = rbtree_Delete (&tree,
    | keyToRemove);
48474| if ( node ) {
48475|     ULONG identifier = 0;
48476|     status = decodeNumbersFromNode (node->Key,
    | node->Pos, firstCluster, numClusters, identifier);
48477|     FreeNode(node);
48478| } else {
48479|     // This should never happen if public callers
    | are working right.
48480|     // Because this member function is protected,
    | only member functions within
48481|     // tVirginMap will call it, and they are all
    | supposed to determine that
48482|     // 'keyToRemove' refers to a node that is
    | definitely within the tree.
48483|     status = STATUS_UNSUCCESSFUL;
48484|     ASSERT(FALSE);
48485| }
48486| return status;
48487| }
48488|
48489| //-----
    | -----
48490|
48491| NTSTATUS tVirginMap::findNextIdentifierForExtentLength
    | (
48492|     CLUSTER_INDEX numClusters,
48493|     CLUSTER_INDEX firstCluster,
48494|     ULONG          &identifier )
48495| {
48496|
    | Profile("tVirginMap::findNextIdentifierForExtentLength")
    | ;
48497|     identifier = MAX_IDENTIFIER;
48498|     keyType key = 0;
48499|     ULONG pos = 0;
48500|     NTSTATUS status = encodeNumbersIntoNode (key, pos,
    | 0, numClusters, 0);
48501|     if ( NT_SUCCESS(status) ) {
48502|         CLUSTER_INDEX numClustersFound = 0;
48503|         CLUSTER_INDEX firstClusterFound = 0;
48504|         ULONG identifierFound = 0;
48505|         tTreeLeaf *node = rbtree_SearchUpperBound
    | (&tree, key);
48506|         if ( node ) {
48507|             status = decodeNumbersFromNode (node->Key,
    | node->Pos, firstClusterFound, numClustersFound,

```

```

    | identifierFound);
48508|         if ( NT_SUCCESS(status) &&
    | numClustersFound==numClusters ) {
48509|             // If identifierFound==0, it means we
    | have run out of identifiers for this
48510|             // number of clusters. Just AND with
    | MAX_IDENTIFIER to keep the value within
48511|             // the allowed range. In the
    | wraparound case, chances are the insertion about
48512|             // to happen will fail because of
    | duplicate key, but there are two cases where
48513|             // this will not happen: either the 8
    | high bits of the start cluster being
48514|             // different, or the extent that
    | already has id==MAX_IDENTIFIER having already
48515|             // been deleted due to discovered
    | fragmentation.
48516|             identifier = (identifierFound - 1) &
    | (ULONG)(MAX_IDENTIFIER);
48517|
48518|             // Now check to make sure that this
    | node and all others with the same number of clusters
    | encoded
48519|             // have a different value for the start
    | cluster. If we find any node with the same pair
48520|             // (numClusters,firstCluster), it means
    | that the node about to be inserted doesn't make sense.
48521|             // The reason for doing this is that it
    | indicates a good chance that the caller is in an
48522|             // infinite loop (as is happening right
    | now as I debug pd::FindVirginSpace).
48523|
48524|             while ( node ) {
48525|                 if ( firstClusterFound ==
    | firstCluster ) {
48526|                     status =
    | STATUS_DUPLICATE_OBJECTID;
48527|
    | Debug(DEBUG_DICT,("tVirginMap::findNextIdentifierForExte
    | ntLength: !!! Duplicate found: firstCluster=%08x,
    | numClusters=%08x\n",firstCluster,numClusters));
48528|                     ASSERT(FALSE);
48529|                     break;
48530|                 }
48531|
48532|                 node =
    | rbtree_GetNextInOrder(&tree,node);
48533|                 if ( node ) {
48534|                     status = decodeNumbersFromNode
    | (node->Key, node->Pos, firstClusterFound,

```

```

    | numClustersFound, identifierFound);
48535|         if ( !NT_SUCCESS(status) ) {
48536|             break;
48537|         }
48538|
48539|         if ( numClustersFound !=
    | numClusters ) {
48540|             break;
48541|         }
48542|     }
48543| }
48544| }
48545| }
48546| } else {
48547|     ASSERT(FALSE); // this should never happen
48548| }
48549| return status;
48550| }
48551|
48552| //-----
    | -----
48553|
48554| NTSTATUS tVirginMap::decodeNumbersFromNode (
48555|     keyType     key,
48556|     ULONG       pos,
48557|     CLUSTER_INDEX &firstCluster,
48558|     CLUSTER_INDEX &numClusters,
48559|     ULONG       &identifier ) const
48560| {
48561|     Profile("tVirginMap::decodeNumbersFromNode");
48562|     numClusters = CLUSTER_INDEX(1) + ((key >>
    | (BITS_IN_KEY - BITS_PER_CLUSTER)) &
    | (MAX_VIRGIN_CLUSTERS - 1));
48563|     identifier = ULONG((key >> (BITS_IN_KEY -
    | BITS_PER_CLUSTER - BITS_PER_ID)) & ((1 << BITS_PER_ID)
    | - 1));
48564|     firstCluster = (key & ((keyType(1) <<
    | (BITS_PER_CLUSTER - BITS_IN_POS)) - 1)) << BITS_IN_POS;
48565|     firstCluster |= keyType(pos);
48566|     NTSTATUS status = validateExtent (firstCluster,
    | numClusters);
48567|     return status;
48568| }
48569|
48570| //-----
    | -----
48571|
48572| NTSTATUS tVirginMap::encodeNumbersIntoNode (
48573|     keyType     &key,
48574|     ULONG       &pos,

```

```

48575|  CLUSTER_INDEX  firstCluster,
48576|  CLUSTER_INDEX  numClusters,
48577|  ULONG          identifier ) const
48578| {
48579|  Profile("tVirginMap::encodeNumbersIntoNode");
48580|  key = 0;
48581|  pos = 0;
48582|  NTSTATUS status = validateExtent(firstCluster,
    | numClusters);
48583|  if ( NT_SUCCESS(status) ) {
48584|      key = keyType(numClusters-1) << (BITS_IN_KEY -
    | BITS_PER_CLUSTER);
48585|      key |= keyType(identifier) << (BITS_IN_KEY -
    | BITS_PER_CLUSTER - BITS_PER_ID);
48586|      key |= keyType(firstCluster >> BITS_IN_POS) &
    | ((keyType(1)<<(BITS_PER_CLUSTER - BITS_IN_POS)) - 1);
48587|      pos = unsigned(firstCluster) & ((1u <<
    | BITS_IN_POS) - 1);
48588|  }
48589|
48590| #ifdef DEBUG
48591|  if ( NT_SUCCESS(status) ) {
48592|      // Self-debugging code: decode the numbers back
    | and make sure they match
48593|      // the inputs to the encoder. Initialize the
    | verification values to
48594|      // something that is definitely not the same as
    | the inputs, to catch
48595|      // references not being set by the decoder.
48596|      CLUSTER_INDEX  verifyFirstCluster =
    | ~firstCluster;
48597|      CLUSTER_INDEX  verifyNumClusters =
    | ~numClusters;
48598|      ULONG          verifyIdentifier =
    | ~identifier;
48599|
48600|      NTSTATUS verifyStatus = decodeNumbersFromNode
    | (key, pos, verifyFirstCluster, verifyNumClusters,
    | verifyIdentifier);
48601|      if ( verifyStatus!=STATUS_SUCCESS ||
    | verifyFirstCluster!=firstCluster ||
    | verifyNumClusters!=numClusters ||
    | verifyIdentifier!=identifier ) {
48602|
    | Debug(DEBUG_DICT,("tVirginMap::encodeNumbersIntoNode:\n"
    | ));
48603|      Debug(DEBUG_DICT,("  key=%016l64x,
    | pos=%08x\n",key,pos));
48604|      Debug(DEBUG_DICT,("
    | verifyStatus=%08x\n",verifyStatus));

```

```

48605|         Debug(DEBUG_DICT,("
| firstClusterIn=%016l64x,
| firstClusterOut=%016l64x\n",firstCluster,verifyFirstClus
| ter));
48606|         Debug(DEBUG_DICT,("  numClustersIn=
| %016l64x, numClustersOut=
| %016l64x\n",numClusters,verifyNumClusters));
48607|         Debug(DEBUG_DICT,("  identifierIn= %16x,
| identifierOut= %16x\n",identifier,verifyIdentifier));
48608|
48609|         ASSERT (verifyStatus == STATUS_SUCCESS);
48610|         ASSERT (verifyFirstCluster ==
| firstCluster);
48611|         ASSERT (verifyNumClusters == numClusters);
48612|         ASSERT (verifyIdentifier == identifier);
48613|     }
48614| }
48615| #endif /*DEBUG*/
48616|
48617| return status;
48618| }
48619|
48620| //-----
| -----
48621| void tVirginMap::debugDump()
48622| {
48623| #ifdef DEBUG
48624|     Debug(DEBUG_DICT,("tVirginMap::debugDump():
| this=%08x\n",this));
48625|     ULONG NumNodes = 0;
48626|     if ( tree.HeadLeaf != tree.TailLeaf ) {
48627|         NumNodes = debugDumpRecursive (tree.HeadLeaf,
| 0);
48628|     }
48629|     Debug(DEBUG_DICT,("tVirginMap::debugDump(): number
| of nodes = %08x\n",NumNodes));
48630| #endif /*DEBUG*/
48631| }
48632|
48633| //-----
| -----
48634|
48635| #ifdef DEBUG
48636| ULONG tVirginMap::debugDumpRecursive ( tTreeLeaf *node,
| ULONG depth )
48637| {
48638|     ULONG NumNodes = 1;
48639|     static CLUSTER_INDEX PrevNumClusters = 0;
48640|
48641|     if ( depth == 0 ) {

```

```

48642|    // We are starting to dump the tree. Time to
    | initialize PrevNumClusters to the smallest possible
    | value.
48643|    PrevNumClusters = 0;
48644| }
48645|
48646| if ( depth < 32 ) {
48647|     if ( node->Left != tree.TailLeaf ) {
48648|         NumNodes += debugDumpRecursive (node->Left,
    | 1+depth);
48649|     }
48650|
48651|     CLUSTER_INDEX firstCluster = 0;
48652|     CLUSTER_INDEX numClusters = 0;
48653|     ULONG identifier = 0;
48654|     NTSTATUS decodeStatus = decodeNumbersFromNode
    | (node->Key, node->Pos, firstCluster, numClusters,
    | identifier);
48655|     if ( NT_SUCCESS(decodeStatus) ) {
48656|         Debug(DEBUG_DICT,("  depth=%02x:
    | firstCluster=%016l64x, numClusters=%016l64x,
    | identifier=%08x\n",depth,firstCluster,numClusters,identi
    | fier));
48657|         if ( numClusters < PrevNumClusters ) {
48658|             Debug(DEBUG_DICT,("    ??? Why did
    | numClusters decrease ???\n"));
48659|             ASSERT(FALSE);
48660|         }
48661|         PrevNumClusters = numClusters;
48662|     } else {
48663|         Debug(DEBUG_DICT,("!!! depth=%02x: INVALID
    | - key=%016l64x,
    | pos=%08x\n",depth,node->Key,node->Pos));
48664|     }
48665|
48666|     if ( node->Right != tree.TailLeaf ) {
48667|         NumNodes += debugDumpRecursive
    | (node->Right, 1+depth);
48668|     }
48669| } else {
48670|     ASSERT(FALSE);
48671| }
48672|
48673| return NumNodes;
48674| }
48675| #endif /*DEBUG*/
48676|
48677| //-----
    | -----
48678|

```

```

48679| /*--- end of file iodirect.cpp ---*/
48680|
48681|
48682|
48683| File Listing: iodirect.h
48684|
48685| #define PSM_DIRECTIO_FLAG_ALWAYS_DO_DIRECT 1
48686|
48687| extern ULONG PSMDirectIOOptions;
48688|
48689| const ULONG   RBTREE_KEY_BITS   = 64;
48690| const ULONG   RBTREE_POS_BITS   = 31;
48691|
48692| const ULONG   IODIRECT_BITS_PER_CLUSTER_INDEX   =
    | 39;
48693|
48694| const ULONG   IODIRECT_BITS_PER_CLUSTER_RUN     =
48695|   (RBTREE_KEY_BITS + RBTREE_POS_BITS) -
    | 2*IODIRECT_BITS_PER_CLUSTER_INDEX;
48696|
48697| const ULONG   IODIRECT_MAX_CLUSTER_RUN          = 1
    | << IODIRECT_BITS_PER_CLUSTER_RUN;
48698|
48699| /*-----
    | -----
48700|   We are solving two problems here:
48701|
48702|   1. Given a cluster offset into a file, determine
    | its location on
48703|       disk so we can write directly to the disk.
48704|       This is done with class MapFileToDisk.
48705|
48706|   2. Given a cluster location on disk, determine
    | yes/no whether
48707|       it resides within a PSM file.
48708|       This is done with class MapDiskToFiles.
48709| -----
    | -----*/
48710|
48711| typedef struct sClusterRun {
48712|   ULARGE_INTEGER   FirstCluster;
48713|   ULONG            NumClusters;
48714| } tClusterRun, *pClusterRun;
48715|
48716| //-----
    | -----
48717|
48718| class MapDirectio: public DeletableObject
48719| {
48720| public:

```



```

48721| MapDirectIo() { rbtree_Init(&tree); }
48722| ~MapDirectIo() { reset(); }
48723|
48724| void reset();
48725|
48726| protected:
48727| tTree tree;
48728| };
48729|
48730| //-----
    | -----
48731|
48732| struct RETRIEVAL_POINTERS_BUFFER;
48733| class MapFileToDisk: public MapDirectIo
48734| {
48735| public:
48736| MapFileToDisk();
48737| ~MapFileToDisk();
48738|
48739| NTSTATUS SetInternal (
48740| PDEVICE_OBJECT Volume,
48741| PFILE_OBJECT FileObject,
48742| RETRIEVAL_POINTERS_BUFFER *rp,
48743| ULONG ClusterSizeInBytes,
48744| ULONG SectorSizeInBytes,
48745| bool ReverseMapping );
48746|
48747| ULONG getClusterSizeInBytes() const { return
    | clusterSizeInBytes; }
48748| ULONG getSectorSizeInBytes() const { return
    | sectorSizeInBytes; }
48749|
48750| PDEVICE_OBJECT GetDeviceObject() const { return
    | deviceObject; }
48751|
48752| NTSTATUS getDiskOffsetForFileOffset (
48753| LARGE_INTEGER FileOffsetInBytes,
48754| LARGE_INTEGER &DiskOffsetInBytes,
48755| LARGE_INTEGER &ExtentLengthInBytes,
48756| bool SuppressDebugErrors );
48757|
48758| void close();
48759| void setInverseMapFlag() { inverseMapFlag = true; }
48760|
48761| protected:
48762| NTSTATUS insertExtent (
48763| LARGE_INTEGER FileClusterOffset,
48764| LARGE_INTEGER DiskClusterOffset,
48765| LARGE_INTEGER NumClusters );
48766|

```

```

48767| private:
48768|     bool        openFlag;
48769|     bool        inverseMapFlag;
48770|     ULONG        clusterSizeInBytes;
48771|     ULONG        sectorSizeInBytes;
48772|     PFILE_OBJECT fileObject;
48773|     PDEVICE_OBJECT deviceObject;
48774| };
48775|
48776| //-----
| -----
48777| // class DirectAccessFile
48778| //
48779| // Allows reads and writes directly to a file without
| using the filesystem.
48780| //
48781| // Must open the file while the filesystem is still
| mounted.
48782| // Can read, write, and close with or without the
| filesystem.
48783| // All I/O must be done in integer multiples of
| sectors.
48784| //
48785| // NOTE: When writing, cannot change the size of the
| file.
48786| //     Can only modify the data that's already
| there!
48787| //-----
| -----
48788| class DirectAccessFile: public DeletableObject
48789| {
48790| public:
48791|     DirectAccessFile(): isOpen(false)
48792|     {
48793|         diskMap.setInverseMapFlag();
48794|     }
48795|
48796|     ~DirectAccessFile() { close(); }
48797|
48798| // NTSTATUS open ( WCHAR *FileName );
48799|
48800|
48801|     bool readyForDirectIo() const { return isOpen; }
48802|
48803|     NTSTATUS SetInternal (
48804|         PDEVICE_OBJECT      Volume,
48805|         PFILE_OBJECT        FileObject,
48806|         RETRIEVAL_POINTERS_BUFFER *rp,
48807|         ULONG                ClusterSizeInBytes,
48808|         ULONG                SectorSizeInBytes

```

```

| );
48809|
48810| NTSTATUS read (
48811|     PVOID      Buffer,          // where to
| put the data
48812|     LARGE_INTEGER FileOffsetInBytes, // must be
| multiple of sector size
48813|     ULONG      BytesToRead );    // must be
| multiple of sector size
48814|
48815| NTSTATUS write (
48816|     PCVOID      Buffer,          // data to
| write
48817|     LARGE_INTEGER FileOffsetInBytes, // must be
| multiple of sector size
48818|     ULONG      BytesToWrite );    // must be
| multiple of sector size
48819|
48820| void close(); // It cannot fail! HAH!
48821|
48822| ULONG getClusterSizeInBytes() const { return
| fileMap.getClusterSizeInBytes(); }
48823| ULONG getSectorSizeInBytes() const { return
| fileMap.getSectorSizeInBytes(); }
48824|
48825| NTSTATUS getFileOffset (
48826|     LARGE_INTEGER DiskOffsetInBytes,
48827|     LARGE_INTEGER &FileOffsetInBytes,
48828|     LARGE_INTEGER &ExtentLengthInBytes );
48829|
48830| NTSTATUS getFileSizeInBytes ( LARGE_INTEGER
| &SizeInBytes );
48831|
48832| protected:
48833| enum IOTYPE { IOTYPE_UNDEFINED=0, IOTYPE_READ=1,
| IOTYPE_WRITE=2};
48834|
48835| NTSTATUS perform_io (
48836|     PVOID      Buffer,
48837|     LARGE_INTEGER FileOffsetInBytes,
48838|     ULONG      BytesToProcess,
48839|     IOTYPE      IoType,
48840|     const char  *IoTypeString );
48841|
48842| private:
48843|     bool      isOpen;
48844|     MapFileToDisk fileMap;
48845|     MapFileToDisk diskMap;
48846| };
48847|

```

```

48848|
48849| //-----
48850| | -----
48851| #ifdef DEBUG
48852|     void TestDirectRead();
48853| #endif
48854|
48855| typedef __int64 CLUSTER_INDEX;
48856|
48857| class tVirginMap: public MapDirectlo
48858| {
48859| public:
48860|     tVirginMap():
48861|         MapDirectlo(),
48862|         totalClustersOnVolume(0)
48863|     {}
48864|
48865|     bool isEmpty() const;
48866|
48867|     NTSTATUS insertExtent (
48868|         CLUSTER_INDEX firstCluster,
48869|         CLUSTER_INDEX numClusters );
48870|
48871|     NTSTATUS removeAnyExtent (
48872|         CLUSTER_INDEX &firstCluster,
48873|         CLUSTER_INDEX &numClusters );
48874|
48875|     NTSTATUS removeLongestExtent (
48876|         CLUSTER_INDEX &firstCluster,
48877|         CLUSTER_INDEX &numClusters );
48878|
48879|     void debugDump();
48880|
48881|     NTSTATUS queryLongestExtent ( CLUSTER_INDEX
48882|         | &numClusters ) const;
48883|
48884|     void enforceClusterLimit ( CLUSTER_INDEX
48885|         | _totalClustersOnVolume ) { totalClustersOnVolume =
48886|         | _totalClustersOnVolume; }
48887|
48888|     CLUSTER_INDEX queryTotalClusters() const { return
48889|         | totalClustersOnVolume; }
48890|
48891|     NTSTATUS validateExtent (
48892|         CLUSTER_INDEX firstCluster,
48893|         CLUSTER_INDEX numClusters ) const;
48894|
48895| protected:
48896|     NTSTATUS findNextIdentifierForExtentLength (
48897|         | CLUSTER_INDEX numClusters, CLUSTER_INDEX firstCluster,

```

```

    | ULONG &NextIdentifier );
48892|
48893| NTSTATUS removeExtentByKey (
48894|     keyType      keyToRemove,
48895|     CLUSTER_INDEX &firstCluster,
48896|     CLUSTER_INDEX &numClusters );
48897|
48898| #ifdef DEBUG
48899|     ULONG debugDumpRecursive ( tTreeLeaf *node, ULONG
    | depth );
48900| #endif /*DEBUG*/
48901|
48902| protected:
48903| NTSTATUS decodeNumbersFromNode (
48904|     keyType      key,
48905|     ULONG        pos,
48906|     CLUSTER_INDEX &firstCluster,
48907|     CLUSTER_INDEX &numClusters,
48908|     ULONG        &identifier ) const;
48909|
48910| NTSTATUS encodeNumbersIntoNode (
48911|     keyType      &key,
48912|     ULONG        &pos,
48913|     CLUSTER_INDEX firstCluster,
48914|     CLUSTER_INDEX numClusters,
48915|     ULONG        identifier ) const;
48916|
48917| private:
48918|     CLUSTER_INDEX totalClustersOnVolume;    // if
    | not zero, indicates total number of clusters on volume
48919| };
48920|
48921| #ifdef DEBUG
48922| void TestVirginMap(void *);
48923| #endif /*DEBUG*/
48924|
48925| /*--- end of file iodirect.h ---*/
48926|
48927|
48928|
48929| File Listing: IRP.cpp
48930|
48931| #include "precomp.h"
48932|
48933| #ifndef IRP_DEBUG
48934| #define MAX_STACK_LOCATIONS 32
48935| STATIC LIST_ENTRY IrpFreeList[MAX_STACK_LOCATIONS]={0};
48936| STATIC KSPIN_LOCK IrpFreeListLock={0};
48937| STATIC ULONG      IrpInited=0;
48938| #endif

```

```

48939|
48940|
48941|
48942| /*-----
    | -----*/
48943| VOID IrpInit( VOID ) {
48944| #ifndef IRP_DEBUG
48945|     int i;
48946|
48947|     for (i=0;i<MAX_STACK_LOCATIONS;i++)
48948|         InitializeListHead(&IrpFreeList[i]);
48949|
48950|     KeInitializeSpinLock(&IrpFreeListLock);
48951|
    | pmRegisterObject(&IrpFreeListLock,"IrpFreeListLock",pmSp
    | inLock);
48952|     IrpInited = TRUE;
48953| #endif
48954|     return;
48955| }
48956|
48957| /*-----
    | -----*/
48958| VOID IrpDeInit( VOID ) {
48959| #ifndef IRP_DEBUG
48960|     int i;
48961|     KIRQL oldIrql;
48962|
48963|     pmAcquireSpinLock(&IrpFreeListLock, &oldIrql);
48964|
48965|     for (i=0;i<MAX_STACK_LOCATIONS;i++) {
48966|
48967|         while (!IsListEmpty(&IrpFreeList[i])) {
48968|             PLIST_ENTRY listEntry;
48969|             PIRP     newIrp;
48970|
48971|             // Remove the head of the list.
48972|             listEntry =
    | RemoveHeadList(&IrpFreeList[i]);
48973|
48974|             if((listEntry) &&
    | (listEntry!=&IrpFreeList[i])) {
48975|                 // Convert into an IRP pointer.
48976|
48977|                 /*lint -save -e413 */
48978|                 newIrp = CONTAINING_RECORD(listEntry,
    | IRP, Tail.Overlay.ListEntry);
48979|                 /*lint -restore */
48980|                 FREE_POINTER( newIrp );
48981|             } else {

```

```

48982|         Debug(DEBUG_INIT,("IrpDelnit: ListEntry
    | is empty\n"));
48983|     }
48984| }
48985| }
48986|
48987| IrpInited = FALSE;
48988| pmReleaseSpinLock(&IrpFreeListLock, oldIrq);
48989| pmDeRegisterObject(&IrpFreeListLock);
48990| #endif
48991| return;
48992| }
48993|
48994| /*-----
    | -----*/
48995| PIRP IrpAllocateIrp( ULONG StackSize ) {
48996| #ifndef IRP_DEBUG
48997|     KIRQL oldIrq;
48998|     PIRP newIrp = NULL;
48999|
49000|     ASSERT(StackSize < MAX_STACK_LOCATIONS);
49001|
49002|     if((IrpInited) && (StackSize<MAX_STACK_LOCATIONS))
        | {
49003|
49004|         pmAcquireSpinLock(&IrpFreeListLock, &oldIrq);
49005|
49006|         if(IrpInited) {
49007|             if (!IsListEmpty(&IrpFreeList[StackSize]))
                | {
49008|                 PLIST_ENTRY listEntry;
49009|
49010|                 // Remove the head of the list.
49011|                 listEntry =
                    | RemoveHeadList(&IrpFreeList[StackSize]);
49012|
49013|                 if((listEntry) &&
                    | (listEntry!=&IrpFreeList[StackSize])) {
49014|                     // Convert into an IRP pointer.
49015|                     /*lint -save -e413 */
49016|                     newIrp =
                        | CONTAINING_RECORD(listEntry, IRP,
                        | Tail.Overlay.ListEntry);
49017|                     /*lint -restore */
49018|                 } else {
49019|                     Debug(DEBUG_INIT,("IrpAllocateIrp:
                        | ListEntry is empty\n"));
49020|                 }
49021|             }
49022|         } else {

```

```

49023|         Debug(DEBUG_INIT,("IrpAllocatelp:
| IrpInited set to false after we acquired spinlock\n"));
49024|         ASSERT(FALSE);
49025|     }
49026|
49027|     pmReleaseSpinLock(&IrpFreeListLock, oldIrql);
49028| }
49029|
49030| // if none on the list, allocate one
49031| if (!newIrp) {
49032|     newIrp = (PIRP)
| MemAllocatePoolWithTag(NonPagedPool,
| IoSizeOfIrp(StackSize), IRPTAG);
49033|
49034|     /*
49035|     At this point, we could allocate from the
| XXXMustSucceed pool, but
49036|     it is limited.. instead, we will just return
| null, and eventually,
49037|     the backup will be aborted (The higher level
| code checks for NULL)
49038|     */
49039| }
49040|
49041| // init the irp (if there is one)
49042| if(newIrp) {
49043|     IoInitializeIrp(newIrp, IoSizeOfIrp(StackSize),
| (CHAR)StackSize);
49044| }
49045|
49046| return newIrp;
49047| #else
49048| return IoAllocateIrp((CHAR)StackSize & 0xff,FALSE);
49049| #endif
49050| }
49051|
49052| /*-----*/
| -----*/
49053| VOID IrpFreeIrp( PIRP Irp ) {
49054|     KIRQL oldIrql;
49055|     ASSERT(Irp); //
| not NULL
49056|     ASSERT(Irp->Type == IO_TYPE_IRP); //
| is an IRP
49057|
49058| #ifndef IRP_DEBUG
49059|     ASSERT(Irp->StackCount < MAX_STACK_LOCATIONS); //
| Not greater than we can handle
49060| // ASSERT(Irp->CurrentLocation<=Irp->StackCount);
| // no overflow..

```



```

49061|    // init it so we can tell its free when using the
      | debugger
49062|    // but this is really redundant as the allocIrp
      | code does it also
49063| #ifdef DEBUG
49064|    // verify Irp is not corrupt since we never free
      | them
49065|    #if MEMDBG
49066|        /*lint -save -e522 */
49067|        MemCheckPool(Irp);
49068|        /*lint -restore */
49069|    #endif
49070|
49071|    IoInitializeIrp(Irp, IoSizeOfIrp(Irp->StackCount),
      | Irp->StackCount);
49072| #endif
49073|
49074|    pmAcquireSpinLock(&IrpFreeListLock, &oldIrql);
49075|    if((IrpInited) &&
      | (Irp->StackCount<MAX_STACK_LOCATIONS)) {
49076|        // put it on the free list.
49077|
      | InsertTailList(&IrpFreeList[Irp->StackCount],&Irp->Tail.
      | Overlay.ListEntry);
49078|        pmReleaseSpinLock(&IrpFreeListLock, oldIrql);
49079|    } else {
49080|        pmReleaseSpinLock(&IrpFreeListLock, oldIrql);
49081|        MemFreePool(Irp);
49082|    }
49083| #else
49084|    IoFreeIrp(Irp);
49085| #endif
49086|    return;
49087| }
49088|
49089|
49090|
49091| File Listing: IRP.h
49092|
49093| VOID IrpInit( VOID );
49094| PIRP IrpAllocateIrp( ULONG StackSize );
49095| VOID IrpFreeIrp( PIRP Irp );
49096| VOID IrpDelInit( VOID );
49097|
49098|
49099|
49100| File Listing: LOG.cpp
49101|
49102| #include "precomp.h"
49103|

```

```

49104|
49105| NTSTATUS LogError( PDEVICE_OBJECT DeviceObject,
49106|                 PIRP Irp,
49107|                 ULONG MessageId,
49108|                 NTSTATUS Error,
49109|                 PVOID DumpData,
49110|                 ULONG DumpDataSize, // in bytes!!!
49111|                 WCHAR *Strings[],
49112|                 ULONG NumStrings )
49113| {
49114|     PIO_ERROR_LOG_PACKET errorLogEntry=NULL;
49115|     PIO_ERROR_LOG_PACKET LogEntry=NULL;
49116|     PWCHAR insertionString=NULL;
49117|     USHORT Strlens=0;
49118|     ULONG i;
49119|     NTSTATUS Status=STATUS_SUCCESS;
49120|     USHORT LogSize=0;
49121|     USHORT SafeDumpSize=0;
49122|     USHORT DumpOffset=0;
49123|
49124|     for(i=0;i<NumStrings;i++) {
49125|         Strlens+=NumBytes(Strings[i])+sizeof(WCHAR); //
            | add in null character
49126|     }
49127|
49128|     // round up to nearest dword length
49129|     SafeDumpSize = ((USHORT)((DumpDataSize+3) /
            | sizeof(ULONG)) * sizeof(ULONG));
49130|
49131| /*
49132| typedef struct _IO_ERROR_LOG_PACKET {
49133|     UCHAR MajorFunctionCode;           //0
49134|     UCHAR RetryCount;                  1
49135|     USHORT DumpDataSize;               2
49136|     USHORT NumberOfStrings;            4
49137|     USHORT StringOffset;               6
49138|     USHORT EventCategory;              8
49139|     NTSTATUS ErrorCode;                c
49140|     ULONG UniqueErrorValue;            10
49141|     NTSTATUS FinalStatus;              14
49142|     ULONG SequenceNumber;              18
49143|     ULONG IoControlCode;               1c
49144|     LARGE_INTEGER DeviceOffset;        20
49145|     ULONG DumpData[1];                 28
49146|                                     2c
49147| }IO_ERROR_LOG_PACKET, *PIO_ERROR_LOG_PACKET;
49148| */
49149|
49150| /*lint -save -e734 -e545 */
49151|     DumpOffset =

```

```

    | (USHORT)FIELD_OFFSET(IO_ERROR_LOG_PACKET,DumpData);
49152|  /*lint -restore */
49153|
49154|  LogSize = DumpOffset;    // Have it start
    | here(28) instead of after (30)
49155|  LogSize += SafeDumpSize; // number of bytes
    | needed rounded to dword
49156|  LogSize += Strlens;      // length of strings
49157|
49158|  // can get bigger than a byte (255)
49159|  ASSERT(LogSize<=ERROR_LOG_MAXIMUM_SIZE);
49160|
49161|  // make sure at least as big as structure (its
    | aligned funny)
49162|  if(LogSize<sizeof(IO_ERROR_LOG_PACKET)) {
49163|      LogSize = sizeof(IO_ERROR_LOG_PACKET);
49164|  }
49165|
49166|  errorLogEntry = (PIO_ERROR_LOG_PACKET)
    | MemAllocatePoolWithTag( PagedPool,LogSize,EVENTTAG);
49167|
49168|  if(errorLogEntry) {
49169|      // mandatory
49170|      errorLogEntry->ErrorCode = MessageId;
49171|
49172|      // optional
49173|      errorLogEntry->StringOffset = NumStrings>0 ?
    | DumpOffset+(USHORT)(SafeDumpSize) : 0;
49174|      errorLogEntry->NumberOfStrings =
    | (USHORT)NumStrings;
49175|      errorLogEntry->DumpDataSize =
    | (USHORT)SafeDumpSize;
49176|      errorLogEntry->SequenceNumber = 0;
49177|      errorLogEntry->RetryCount = 0;
49178|      errorLogEntry->UniqueErrorValue = 0;
49179|      errorLogEntry->FinalStatus = Error;
49180|      errorLogEntry->EventCategory = 0;
49181|
49182|      if(Irp) {
49183|          PIO_STACK_LOCATION IrpStack =
    | IoGetCurrentIrpStackLocation(Irp);
49184|
49185|          errorLogEntry->MajorFunctionCode =
    | IrpStack->MajorFunction;
49186|
49187|          if( (IrpStack->MajorFunction ==
    | IRP_MJ_DEVICE_CONTROL) ||
49188|              (IrpStack->MajorFunction ==
    | IRP_MJ_INTERNAL_DEVICE_CONTROL) ||
49189|              (IrpStack->MajorFunction ==

```

```

    | IRP_MJ_SCSI) ) {
49190|
49191|         errorLogEntry->IoControlCode =
    | IrpStack->Parameters.DeviceIoControl.IoControlCode;
49192|         errorLogEntry->DeviceOffset.QuadPart =
    | 0;
49193|     } else {
49194|         errorLogEntry->DeviceOffset =
    | IrpStack->Parameters.Read.ByteOffset;
49195|         errorLogEntry->IoControlCode = 0;
49196|     }
49197| } else {
49198|     errorLogEntry->MajorFunctionCode = 0 ;
49199|     errorLogEntry->IoControlCode = 0;
49200|     errorLogEntry->DeviceOffset.QuadPart = 0;
49201| }
49202|
49203| if(DumpDataSize>0) {
49204|     | RtlZeroMemory(errorLogEntry->DumpData, SafeDumpSize);
49205|     | RtlMoveMemory(errorLogEntry->DumpData, DumpData, DumpDataS
    | ize);
49206| }
49207|
49208|     insertionString =
    | (PWSTR)((PCHAR)(errorLogEntry) +
    | errorLogEntry->StringOffset);
49209|
49210|     for(i=0; i<NumStrings; i++) {
49211|         RtlCopyMemory(insertionString, Strings[i],
    | NumBytes(Strings[i])+sizeof(WCHAR));
49212|         insertionString+=NumChars(Strings[i])+1; //
    | in WCHARs
49213|     }
49214|
49215|     // we do this so we can check we dont overwrite
    | our buffers.
49216|     LogEntry = (PIO_ERROR_LOG_PACKET)
    | IoAllocateErrorLogEntry( DeviceObject, (UCHAR)LogSize);
49217|     if(LogEntry) {
49218|         | RtlMoveMemory(LogEntry, errorLogEntry, LogSize);
49219|         IoWriteErrorLogEntry (LogEntry);
49220|     } else {
49221|         Debug(DEBUG_INIT, ("Error allocating error
    | log entry\n"));
49222|         Status = STATUS_INSUFFICIENT_RESOURCES;
49223|     }
49224|

```

```

49225|     FREE_POINTER(errorLogEntry);
49226|     Status=STATUS_SUCCESS;
49227| } else {
49228|     Debug(DEBUG_INIT,("Error allocating error log
    | entry memory\n"));
49229|     Status = STATUS_INSUFFICIENT_RESOURCES;
49230| }
49231| return Status;
49232| }
49233|
49234|
49235|
49236| File Listing: LOG.h
49237|
49238| NTSTATUS LogError( PDEVICE_OBJECT DeviceObject,
49239|     PIRP Irp,
49240|     ULONG MessageId,
49241|     NTSTATUS Error,
49242|     PVOID DumpData,
49243|     ULONG DumpDataSize, // in bytes
49244|     WCHAR *Strings[],
49245|     ULONG NumStrings );
49246|
49247|
49248|
49249| File Listing: MEM.cpp
49250|
49251| #include "precomp.h"
49252|
49253| ZONE_HEADER Zone;
49254| ULONG ZoneIncrementInNodes = 64*1024;    // if we
    | call AllocNode but there is no memory left in the zone,
    | this is how much we extend by
49255| ULONG ZoneInitialized = FALSE;    // so we
    | know to initialize the zone only once.
49256| ULONG ZoneUsageCount = 0;    // how many
    | entities are referring to the zone (1 for DirectIO,
    | plus 1 for every active volume)
49257| ULONG ZoneMegabytesTotal = 0;    // number
    | of megabytes allocated to zone
49258| ULONG ZoneMegabytesUnused = 0;    // how many
    | megabytes in the zone are sitting idle
49259| ULONG ZonePrivateAccess = 1;    // used in
    | CMPXCHG for pseudo-spinlock
49260| ULONG ZoneLoggedOutOfMemory = FALSE;    // if
    | AllocNode ever reports out-of-memory, it logs an event,
    | but only once.
49261|
49262| const ULONG MIN_RECURSIVE_BLOCK_BYTES = 1024 *
    | 1024;    // smallest number of bytes we will

```

```

    | allocate in MemoryAllocationFunction
49263|
49264| // turn off for testing new zone stuff
49265| #undef _USE_MEM_CHECK_
49266|
49267| /*
49268|    We protect ourselves by using these functions
    | instead of spinlocks.
49269|    This is done because spinlocks change the irq to
    | DISPATCH_LEVEL you
49270|    can not access paged memory at that level. If the
    | memory holding the
49271|    node to add is paged then a bugcheck can occur (it
    | will if driver verifier
49272|    is running on windows 2000).
49273| */
49274|
49275| /*
49276|    Get exclusive access to private region
49277|    IRQL remains the same
49278|    Thread switching will still occur
49279|    Multiple processor safe
49280| */
49281| void MyGetPrivateAccess( PLONG PAccess, ULONG *Save )
49282| {
49283|    ULONG X=0;
49284|
49285|    while( X==0 ) {
49286| #if _WIN32_WINNT>=0x0500
49287|        X = (ULONG)InterlockedCompareExchange(
    | (PLONG)PAccess,0,(LONG)1 );
49288| #else
49289|        X = (ULONG)InterlockedCompareExchange( (PVOID
    | *)PAccess,0,(PVOID)1 );
49290| #endif
49291|        // Give up reset of quantum if we can
49292|        if( (!X) && (KeGetCurrentIrql() <
    | DISPATCH_LEVEL) ) {
49293|            pmThreadSwitch();
49294|        }
49295|    }
49296|    *Save=X;
49297| }
49298|
49299|
49300| void MyReleasePrivateAccess( PLONG PAccess, ULONG Save
    | )
49301| {
49302|    ASSERT(Save!=0);
49303|

```

```

49304|   InterlockedExchange( PAccess, Save );
49305| }
49306|
49307| //-----
| -----
| -----
49308|
49309| struct tNodeBlockHeader {
49310|   tNodeBlockHeader *next;      // pointer
| to next tNodeBlockHeader in list
49311|   tNodeBlockHeader *tail;      // pointer
| to final tNodeBlockHeader in list
49312|   ULONG             sizeInBytes; // size of
| this block in bytes (excluding ZONE_SEGMENT_HEADER)
49313| };
49314|
49315| typedef tNodeBlockHeader *pNodeBlockHeader;
49316|
49317| //-----
| -----
| -----
49318|
49319| void FreeMemoryFunction ( PVOID &NodeBlock )
49320| {
49321|   while ( NodeBlock ) {
49322|     PVOID NextBlock =
| pNodeBlockHeader(NodeBlock)->next;
49323|     FREE_POINTER(NodeBlock);
49324|     NodeBlock = NextBlock;
49325|   }
49326| }
49327|
49328| //-----
| -----
| -----
49329|
49330| #ifdef DEBUG
49331|
49332| int ListLength ( pNodeBlockHeader Node )
49333| {
49334|   int count = 0;
49335|
49336|   ASSERT (Node != NULL);
49337|   if ( Node != NULL ) {
49338|     const pNodeBlockHeader CheckTail = Node->tail;
49339|     ASSERT(CheckTail != NULL);
49340|
49341|     while ( Node ) {
49342|       ++count;
49343|       if ( Node->next == NULL ) {

```

```

49344|         ASSERT (CheckTail == Node);
49345|     }
49346|     Node = Node->next;
49347| }
49348| }
49349|
49350| return count;
49351| }
49352|
49353| #endif /*DEBUG*/
49354|
49355| //-----
| -----
| -----
49356|
49357| static const ULONG ALLOC_MAX_RECURSION_DEPTH = 30;
49358|
49359| NTSTATUS AllocateMemoryFunction (
49360|     ULONG     NumNodesNeeded,
49361|     PVOID     &NodeBlock,
49362|     ULONG     RecursionDepth )
49363| {
49364|     NTSTATUS Status = STATUS_INSUFFICIENT_RESOURCES;
| // assume failure until success is proven
49365|     NodeBlock = NULL;
49366|
49367|     ULONG NumBytesNeeded =
| NumNodesNeeded*sizeof(tTreeLeaf) +
| sizeof(ZONE_SEGMENT_HEADER);
49368|     Debug(DEBUG_MEMORY,("[ZONE] AllocateMemoryFunction:
| NodesNeeded=%08x, BytesNeeded=%08x,
| RecursionDepth=%08x\n",NumNodesNeeded,NumBytesNeeded,Rec
| ursionDepth));
49369|
49370|     if ( NumBytesNeeded < MIN_RECURSIVE_BLOCK_BYTES ) {
49371|         // Recursion Stopper #1: We never allow an
| allocation less than 1MB.
49372|         Status = STATUS_INSUFFICIENT_RESOURCES;
49373|     } else if ( RecursionDepth >
| ALLOC_MAX_RECURSION_DEPTH ) {
49374|         // Recursion Stopper #2: Never let recursion
| get too deep.
49375|         Status = STATUS_INSUFFICIENT_RESOURCES;
49376|         ASSERT (RecursionDepth <=
| ALLOC_MAX_RECURSION_DEPTH);
49377|     } else {
49378|         // try to allocate the block requested
| directly...
49379|         NodeBlock =
| MemAllocatePoolWithTag(PagedPool,NumBytesNeeded,NODEMEMT

```



```

    | AG);
49380|     if ( NodeBlock ) {
49381|         // Recursion Stopper #3: We successfully
    | allocated a contiguous chunk of the desired size!
49382|         // Make the block be a linked list having a
    | length of one node.
49383|         pNodeBlockHeader(NodeBlock)->next
    | = NULL;
49384|         pNodeBlockHeader(NodeBlock)->tail
    | = pNodeBlockHeader(NodeBlock);
49385|         pNodeBlockHeader(NodeBlock)->sizeInBytes
    | = NumBytesNeeded - sizeof(ZONE_SEGMENT_HEADER);    //
    | store space usable for nodes only (for math reasons)
49386|         Status = STATUS_SUCCESS;
49387|     } else {
49388|         // Here's where we need recursion: We
    | could not allocate a contiguous chunk of the desired
    | size.
49389|         // We we will recursively split this
    | request into a request for N/2 and one for (N - N/2),
49390|         // which always adds up to N.
49391|         //
49392|         // Important: Note that if N is odd, then
    | N/2 will be smaller than (N - N/2).
49393|         // Example: If N=7, then N/2=3, and (N -
    | N/2)=4.
49394|         // Also, 3+4=7, so number of nodes is
    | preserved exactly.
49395|         // (If N is even, the two expressions are
    | equal.)
49396|         //
49397|         // We prefer to allocate the smaller of the
    | two first, because if it is too small,
49398|         // the recursive call will return a failure
    | status, and we won't even try to allocate
49399|         // the bigger size.
49400|
49401|         PVOID LeftBlock = NULL;    // first
    | sublist to allocate
49402|         Status = AllocateMemoryFunction (
    | (NumNodesNeeded/2), LeftBlock, 1+RecursionDepth );
49403|         if ( NT_SUCCESS(Status) ) {
49404|             PVOID RightBlock = NULL;    // second
    | sublist to allocate
49405|             Status = AllocateMemoryFunction (
    | NumNodesNeeded-(NumNodesNeeded/2), RightBlock,
    | 1+RecursionDepth );
49406|             if ( NT_SUCCESS(Status) ) {
49407|                 // We successfully allocated both
    | the left and right sublists.

```

```

49408|
49409|         #ifdef DEBUG
49410|         // Remember how many blocks
         | there are in each sublist.
49411|         const int LeftLength  =
         | ListLength(pNodeBlockHeader(LeftBlock));
49412|         const int RightLength =
         | ListLength(pNodeBlockHeader(RightBlock));
49413|         #endif /*DEBUG*/
49414|
49415|         // We need to concatenate LeftBlock
         | list and RightBlock list
49416|         // so as to make NodeBlock a single
         | longer list containing all the nodes
49417|         // of both sublists.
49418|         | pNodeBlockHeader(LeftBlock)->tail->next  =
         | pNodeBlockHeader(RightBlock);
49419|         pNodeBlockHeader(LeftBlock)->tail
         | = pNodeBlockHeader(RightBlock)->tail;
49420|         NodeBlock = LeftBlock;
49421|
49422|         #ifdef DEBUG
49423|         // Make sure the combined list
         | has exactly the right number of nodes...
49424|         const int TotalLength =
         | ListLength(pNodeBlockHeader(NodeBlock));
49425|         ASSERT ( LeftLength +
         | RightLength == TotalLength );
49426|         #endif /*DEBUG*/
49427|     } else {
49428|         // We already allocated the left
         | sublist successfully,
49429|         // but failed in the right sublist.
49430|         // We need to guarantee that every
         | single allocated block
49431|         // of memory is freed. Any time
         | this function returns
49432|         // a failure code, it is guaranteed
         | to not have orphaned
49433|         // any memory. Therefore, all we
         | need to do is to free the
49434|         // left sublist.
49435|
49436|         FreeMemoryFunction (LeftBlock);
49437|     }
49438| }
49439| }
49440| }
49441|

```

```

49442| #ifdef DEBUG
49443|     // Postcondition sanity checks...
49444|     if ( NT_SUCCESS(Status) ) {
49445|         // If we are returning success, we are supposed
         | to have allocated and linked a valid list of blocks...
49446|         ASSERT (NodeBlock != NULL);
49447|         if ( NodeBlock != NULL ) {
49448|             ASSERT (pNodeBlockHeader(NodeBlock)->tail
         | != NULL);
49449|             if ( pNodeBlockHeader(NodeBlock)->tail !=
         | NULL ) {
49450|                 ASSERT
         | (pNodeBlockHeader(NodeBlock)->tail->next == NULL);
49451|             }
49452|         }
49453|     } else {
49454|         // If we are returning failure, we should have
         | left NodeBlock==NULL...
49455|         ASSERT (NodeBlock == NULL);
49456|     }
49457| #endif /*DEBUG*/
49458|
49459|     ASSERT (Status==STATUS_SUCCESS ||
         | Status==STATUS_INSUFFICIENT_RESOURCES);
49460|     Status = NT_SUCCESS(Status) ? STATUS_SUCCESS :
         | STATUS_INSUFFICIENT_RESOURCES;
49461|     return Status;
49462| }
49463|
49464| //-----
         | -----
         | -----
49465|
49466| NTSTATUS ExtendZoneWithNodeBlockList (
49467|     PZONE_HEADER   Zone,
49468|     PVOID          NodeBlockList,
49469|     ULONG          IncrementStepInNodes )
49470| {
49471|     NTSTATUS Status = STATUS_INSUFFICIENT_RESOURCES;
49472|
49473|     Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList:
         | Zone=%08x, NodeBlockList=%08x,
         | IncrementNodes=%08x\n",Zone,NodeBlockList,IncrementStepI
         | nNodes));
49474|
49475|     ULONGLONG TotalBytes = 0;
49476|     ULONG NumNodesInList = 0;
49477|     PVOID Node = NodeBlockList;
49478|     while ( Node ) {
49479|         ++NumNodesInList;

```

```

49480|         tNodeBlockHeader BlockHeader =
| *pNodeBlockHeader(Node);    // make safe copy because
| we will use after inserting into zone, thus destroying
| it.
49481|
49482|         | Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList:
| NodeBlockLoop - Node=%08x, Size=%08x, Next=%08x,
| Tail=%08x\n",
49483|             Node,
49484|             BlockHeader.sizeInBytes,
49485|             BlockHeader.next,
49486|             BlockHeader.tail));
49487|
49488|         if ( ZoneInitialized ) {
49489|             ASSERT(BlockHeader.sizeInBytes >=
| MIN_RECURSIVE_BLOCK_BYTES);
49490|             Status =
| ExExtendZone(Zone,Node,BlockHeader.sizeInBytes +
| sizeof(ZONE_SEGMENT_HEADER));
49491|             if ( !NT_SUCCESS(Status) ) {
49492|                 | Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList:
| Error %08x extending zone\n",Status));
49493|                 ASSERT(NT_SUCCESS(Status));
49494|             }
49495|         } else {
49496|             Status =
| ExInitializeZone(Zone,sizeof(tTreeLeaf),Node,BlockHeader
| .sizeInBytes);
49497|             if ( NT_SUCCESS(Status) ) {
49498|                 ZoneInitialized      = TRUE;
49499|                 ZoneUsageCount      = 0;    //
| will be incremented below
49500|                 ZoneIncrementInNodes =
| IncrementStepInNodes;
49501|
49502|                 | Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList: Just
| initialized Zone\n"));
49503|             } else {
49504|                 /*FreeMemoryFunction (Node);*/    //
| not sure this is wise - may be better to leak memory!
49505|                 | Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList:
| Error %08x initializing zone\n",Status));
49506|                 ASSERT(NT_SUCCESS(Status));
49507|             }
49508|         }
49509|

```

```

49510|     if ( NT_SUCCESS(Status) ) {
49511|         const ULONG NodeBytes =
| BlockHeader.sizeInBytes;
49512|
| Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList:
| NodeBytes=%08x, Remainder=%08x\n",NodeBytes,
| (NodeBytes % sizeof(tTreeLeaf)) ));
49513|         ASSERT (NodeBytes % sizeof(tTreeLeaf) ==
| 0);
49514|         ASSERT ( NodeBytes >=
| MIN_RECURSIVE_BLOCK_BYTES );
49515|         TotalBytes += NodeBytes;
49516|     } else {
49517|         break;
49518|     }
49519|
49520|     Node = BlockHeader.next;
49521| }
49522|
49523| if ( NT_SUCCESS(Status) ) {
49524|
| Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList:
| Finished adding list of %08x blocks -
| TotalBytes=%016l64x\n", NumNodesInList, TotalBytes));
49525|     ULONGLONG NumMegabytes_Long = TotalBytes /
| ULONGLONG(1024*1024);
49526|     ULONG NumMegabytes = ULONG(NumMegabytes_Long &
| 0xffffffff);
49527|
| Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList:
| Bytes=%016l64x, NumMegabytes=%08x,
| NumMegabytes_Long=%016l64x\n",TotalBytes,NumMegabytes,Nu
| mMegabytes_Long));
49528|     ASSERT (NumMegabytes == NumMegabytes_Long);
49529|     ZoneMegabytesTotal += NumMegabytes;
49530|
| Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList: Loop
| finished successfully - %08l64x bytes, %08x MB,
| TotalMB=%08x, IdleMB=%08x\n",
49531|         TotalBytes,
49532|         NumMegabytes,
49533|         ZoneMegabytesTotal,
49534|         ZoneMegabytesUnused));
49535| }
49536|
49537| ASSERT (Status==STATUS_SUCCESS ||
| Status==STATUS_INSUFFICIENT_RESOURCES);
49538| Status = NT_SUCCESS(Status) ? STATUS_SUCCESS :
| STATUS_INSUFFICIENT_RESOURCES;
49539| Debug(DEBUG_MEMORY,("ExtendZoneWithNodeBlockList:

```

```

    | Returning Status=%08x
    | -----\n",Status));
49540|   return Status;
49541| }
49542|
49543| //-----
    | -----
    | -----
49544|
49545| NTSTATUS IncreaseNodeMemory (
49546|   ULONG   AdditionalMegabytesNeeded,
49547|   ULONG   IncrementStepInNodes )
49548| {
49549| #ifdef _USE_MEM_CHECK
49550|   #error Cannot compile IncreaseNodeMemory() with
    | _USE_MEM_CHECK defined
49551| #endif /* _USE_MEM_CHECK */
49552|
49553|   NTSTATUS Status = STATUS_INSUFFICIENT_RESOURCES;
49554|   Debug(DEBUG_MEMORY,("[Zone] IncreaseNodeMemory:
    | AdditionalNeeded=%08x MB, IncrementStepInNodes=%08x
    | nodes, ZoneInitialized=%s\n",
49555|     AdditionalMegabytesNeeded,
49556|     IncrementStepInNodes,
49557|     (ZoneInitialized?"TRUE":"FALSE") ));
49558|
49559|   Debug(DEBUG_MEMORY,("[Zone] IncreaseNodeMemory:
    | ZoneMegabytesTotal=%08x,
    | ZoneMegabytesUnused=%08x\n",ZoneMegabytesTotal,ZoneMegab
    | ytesUnused));
49560|
49561|   __try {
49562|     if ( AdditionalMegabytesNeeded >
    | ZoneMegabytesUnused ) {
49563|       // We need to allocate the difference in
    | megabytes and extend the zone with the new memory.
49564|       ULONG NumMegsToAllocate =
    | AdditionalMegabytesNeeded - ZoneMegabytesUnused;
49565|       Debug(DEBUG_MEMORY,("[Zone]
    | IncreaseNodeMemory: Need to allocate %08x
    | MB\n",NumMegsToAllocate));
49566|
49567|       const ULONGLONG BytesNeeded_Long =
    | ULONGLONG(NumMegsToAllocate) * ULONGLONG(1024*1024);
49568|       const ULONG BytesNeeded =
    | ULONG(BytesNeeded_Long & 0xffffffff);
49569|       ASSERT (BytesNeeded == BytesNeeded_Long);
49570|       const ULONG NodesNeeded = (BytesNeeded +
    | sizeof(tTreeLeaf) - 1) / sizeof(tTreeLeaf);
49571|       Debug(DEBUG_MEMORY,("[Zone]

```

```

    | IncreaseNodeMemory: \n"));
49572|
49573|     PVOID BlockList = NULL;
49574|     Status = AllocateMemoryFunction
    | (NodesNeeded, BlockList, 0);
49575|     if ( NT_SUCCESS(Status) ) {
49576|         Status = ExtendZoneWithNodeBlockList
    | (&Zone, BlockList, IncrementStepInNodes);
49577|         if ( NT_SUCCESS(Status) ) {
49578|             ZoneMegabytesUnused = 0; // there
    | is nothing idle now!
49579|         }
49580|     }
49581| } else {
49582|     // We don't need to allocate more memory.
49583|     // We just need to deduct from our current
    | number of idle megabytes in the zone.
49584|
49585|     ZoneMegabytesUnused -=
    | AdditionalMegabytesNeeded;
49586|     Debug(DEBUG_MEMORY,("[Zone]
    | IncreaseNodeMemory: Subtracted from idle megabyte
    | tally :
    | ZoneMegabytesUnused=%08x\n",ZoneMegabytesUnused));
49587|     Status = STATUS_SUCCESS;
49588| }
49589| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
49590|     NTSTATUS ExceptionCode = GetExceptionCode();
49591|     Debug(DEBUG_DICT,("[Zone] Exception %08x in
    | IncreaseNodeMemory\n",ExceptionCode));
49592|     Status = STATUS_INSUFFICIENT_RESOURCES;
49593| }
49594|
49595| if ( NT_SUCCESS(Status) ) {
49596|     ++ZoneUsageCount;
49597|     Debug(DEBUG_MEMORY,("[Zone] IncreaseNodeMemory:
    | ZoneUsage=%08x, returning SUCCESS\n",ZoneUsageCount));
49598| } else {
49599|     Debug(DEBUG_MEMORY,("[Zone] IncreaseNodeMemory:
    | Status=%08x  !!!!! OUT OF MEMORY
    | !!!!!\n",Status));
49600|     /*lint -save -e740 */
49601|
    | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
    | R_OUT_OF_MEMORY,Status,NULL,0,NULL,0);
49602|     /*lint -restore */
49603|     ASSERT(FALSE);
49604| }
49605|

```

```

49606|  ASSERT (Status==STATUS_SUCCESS ||
      | Status==STATUS_INSUFFICIENT_RESOURCES);
49607|  return Status;
49608| }
49609|
49610| //-----
      | -----
49611| void ReclaimNodeMemory ( ULONG NumMegabytesToReclaim )
49612| {
49613|  // We never actually remove memory blocks from the
      | zone.
49614|  // We just keep track of how many megabytes are
      | theoretically
49615|  // needed for each volume (plus initial allocation
      | for DirectIO).
49616|  // We also keep track of the total number of
      | megabytes in the zone.
49617|
49618|  Debug (DEBUG_MEMORY,("[Zone] ReclaimNodeMemory:
      | Count=%08x, MB To Reclaim=%08x, MB Total=%08x, MB
      | Idle=%08x\n",
49619|      ZoneUsageCount,
49620|      NumMegabytesToReclaim,
49621|      ZoneMegabytesTotal,
49622|      ZoneMegabytesUnused));
49623|
49624|  ASSERT (ZoneUsageCount > 1);
      | // Should never reclaim the first alloc (for DirectIO).
49625|  ASSERT (ZoneMegabytesTotal > 0);
      | // Should never reclaim if there is nothing allocated.
49626|  ASSERT (ZoneMegabytesTotal > ZoneMegabytesUnused);
      | // There should always be some memory claimed.
49627|
49628|  ULONG ZoneMegabytesInUse = ZoneMegabytesTotal -
      | ZoneMegabytesUnused;
49629|  ASSERT (NumMegabytesToReclaim <
      | ZoneMegabytesInUse);    // We should never reclaim
      | more than is currently claimed.
49630|
49631|  if ( ZoneUsageCount > 1 ) {    // never
      | unclaim the last entity using the zone
49632|      --ZoneUsageCount;
49633|      ZoneMegabytesUnused += NumMegabytesToReclaim;
49634|  }
49635| }
49636|
49637| //-----
      | -----
49638| void FreeNode( tTreeLeaf *ptr )
49639| {

```



```

49640| #ifndef _USE_MEM_CHECK_
49641|    // ASSERT(GlobalSystemProcessId ==
    | PsGetCurrentProcess());
49642|    ASSERT(ZoneInitialized);
49643|    ULONG oldIrql;
49644|    MyGetPrivateAccess((PLONG) &ZonePrivateAccess,
    | &oldIrql);
49645|    ExFreeToZone(&Zone,ptr);
49646|    MyReleasePrivateAccess((PLONG)
    | &ZonePrivateAccess,oldIrql);
49647| #else
49648|    MemFreePool(ptr);
49649| #endif
49650| }
49651|
49652| //-----
    | -----
49653| tTreeLeaf *AllocNode()
49654| {
49655| #ifndef _USE_MEM_CHECK_
49656|    // ASSERT(GlobalSystemProcessId ==
    | PsGetCurrentProcess());
49657|
49658|    ULONG oldIrql = 0;
49659|    MyGetPrivateAccess((PLONG) &ZonePrivateAccess,
    | &oldIrql);
49660|
49661|    ULONG Amount = ZoneIncrementInNodes;
49662|    ASSERT(ZoneInitialized);
49663|    pTreeLeaf Node =
    | (pTreeLeaf)ExAllocateFromZone(&Zone);
49664|
49665|    if(!Node) {
49666|        ULONG Size;
49667|        PVOID Mem;
49668|
49669|        /*
49670|            ZoneIncrementInNodes is the number of
    | tTreeLeafs to allocate.
49671|
49672|            We will attempt to allocate the entire
    | increment size.
49673|            If that fails, then we will keep dividing
    | by 2 until
49674|            we get a successful allocation or the
    | number of node bytes
49675|            is less than 64k.
49676|            */
49677|
49678| TryAgain:

```

```

49679|      // dont use Increment directly as we want to
      | retry getting that
49680|      // big of a block the next time, incase memory
      | has freed up.
49681|      Size =
      | Amount*sizeof(tTreeLeaf)+sizeof(ZONE_SEGMENT_HEADER);
49682|
49683|      Mem =
      | MemAllocatePoolWithTag(PagedPool,Size,NODEMEMTAG);
49684|      if(Mem) {
49685|          Debug(DEBUG_MEMORY,("AllocNode: Allocated
      | another Zone %d size\n",Size));
49686|
49687|      | if(NT_SUCCESS(ExExtendZone(&Zone,Mem,Size))) {
49688|          Node =
      | (pTreeLeaf)ExAllocateFromZone(&Zone);
49689|      } else {
49690|          Debug(DEBUG_MEMORY,("AllocNode: Error
      | extending zone!!!!\n"));
49691|          ASSERT(FALSE);
49692|          MemFreePool(Mem);
49693|      }
49694|      } else {
49695|          Debug(DEBUG_MEMORY,("AllocNode: Allocating
      | another Zone %d size failed\n",Size));
49696|
49697|          Amount/=2;
49698|
49699|          if(Amount>=(65536 / sizeof(tTreeLeaf))) {
49700|              goto TryAgain;
49701|          }
49702|
49703|          Debug(DEBUG_MEMORY,("[Zone] AllocNode:
      | Error! Out of memory!!!!\n"));
49704|          ASSERT(FALSE);
49705|      }
49706|  }
49707|
49708|  MyReleasePrivateAccess((PLONG)
      | &ZonePrivateAccess,oldIrql);
49709|
49710|  if ( !Node ) {
49711|      if ( !ZoneLoggedOutOfMemory ) {
49712|          ZoneLoggedOutOfMemory = TRUE;    //
      | because we don't want to log zillions of events!
49713|          /*lint -save -e740 */
49714|
      | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_ERRO
      | R_OUT_OF_MEMORY,PSM_ERROR_OUT_OF_MEMORY,NULL,0,NULL,0);

```

```

49715|      /*lint -restore */
49716|      Debug(DEBUG_MEMORY,("[Zone] AllocNode:
| !!!!! OUT OF MEMORY !!!!!\n"));
49717|      ASSERT(FALSE);
49718|  }
49719| }
49720|
49721| return Node;
49722| #else
49723| return
| (pTreeLeaf)MemAllocatePoolWithTag(PagedPool,sizeof(tTree
| Leaf),NODETAG);
49724| #endif
49725| }
49726|
49727| /*--- end of file mem.cpp ---*/
49728|
49729|
49730|
49731| File Listing: MEM.h
49732|
49733| NTSTATUS IncreaseNodeMemory (
49734|     ULONG   AdditionalMegabytesNeeded,
49735|     ULONG   IncrementStepInNodes );
49736|
49737| void ReclaimNodeMemory ( ULONG NumMegabytesToReclaim );
49738|
49739| void FreeNode ( tTreeLeaf *ptr );
49740| tTreeLeaf *AllocNode();
49741|
49742|
49743|
49744| File Listing: MEMTRACK.cpp
49745|
49746| #include "precomp.h"
49747|
49748| #define MEM_LOOKASIDE  100
49749| #define MEM_LARGE     101
49750| #define MEM_SMALL     102
49751|
49752| #undef __PSM_HEADER_DATA_STRUCTURES
49753| namespace pd {
49754|     #include "header.h"
49755| }
49756|
49757|
49758| #define USE_PAGED_FOR_BUFF 0
49759|
49760| #ifdef DEBUG
49761| //#define DEBUG_MEM

```

```

49762| #endif
49763|
49764| PAGED_LOOKASIDE_LIST InternalSnapShot;
49765| NPAGED_LOOKASIDE_LIST MasterSnapShot;
49766| NPAGED_LOOKASIDE_LIST SnapShotEntry;
49767| PAGED_LOOKASIDE_LIST Node;
49768| NPAGED_LOOKASIDE_LIST ReadOnly;
49769| NPAGED_LOOKASIDE_LIST Shared;
49770| NPAGED_LOOKASIDE_LIST Requests;
49771|
49772|
49773| ULONG MemInitd = FALSE;
49774|
49775| #pragma pack(1)
49776| typedef struct sMemHead {
49777|     ULONG Tag;
49778|     WORD HeaderSize;
49779|     BYTE Type;
49780|     BYTE Reserved;
49781| } tMemHead, *pMemHead;
49782| #pragma pack()
49783|
49784|
49785| #ifdef DEBUG_MEM
49786| PVOID
49787| DebugAlloc (
49788|     IN POOL_TYPE PoolType,
49789|     IN ULONG NumberOfBytes,
49790|     IN ULONG Tag
49791| )
49792| {
49793|     PVOID Parent=0, GrandParent=0;
49794|     RtlGetCallersAddress (&Parent, &GrandParent);
49795|     Debug(DEBUG_MEMORY,("NEWMEM: AllocL: p=%08x gp=%08x
| - pool=%08x, size=%08x, Tag=%08x\n",
49796|         Parent,GrandParent,
49797|         PoolType,NumberOfBytes,Tag));
49798|     return
| ExAllocatePoolWithTag(PoolType,NumberOfBytes,Tag);
49799| }
49800| #define DEBUG_ALLOC DebugAlloc
49801| #endif
49802|
49803|
49804|
49805| #ifndef DEBUG_MEM
49806| #define DEBUG_ALLOC NULL
49807| #endif
49808|
49809| void InitGeneralLookaside (

```

```

49810| PGENERAL_LOOKASIDE Lookaside,
49811| IN PALLOCATE_FUNCTION Allocate,
49812| IN PFREE_FUNCTION Free,
49813| IN ULONG Flags,
49814| IN ULONG Size,
49815| IN ULONG Tag,
49816| IN USHORT Depth,
49817| POOL_TYPE Type
49818| )
49819| {
49820|   RtlZeroMemory(Lookaside,sizeof(GENERAL_LOOKASIDE));
49821|
49822|   Lookaside->Type=Type;
49823|   Lookaside->Tag=Tag;
49824|   Lookaside->Size=Size;
49825|   Lookaside->Depth=Depth;
49826|   if(Allocate) {
49827|       Lookaside->Allocate=Allocate;
49828|   } else {
49829|       Lookaside->Allocate=ExAllocatePoolWithTag;
49830|   }
49831|
49832|   if(Free) {
49833|       Lookaside->Free=Free;
49834|   } else {
49835|       Lookaside->Free=ExFreePool;
49836|   }
49837|
49838|   // note: Lookaside->Lock is a mutex, but it is
         | never used
49839|   // so we will not initialize.
49840|   ExInitializeSListHead(&Lookaside->ListHead);
49841|
49842|   InitializeListHead(&Lookaside->ListEntry);
49843| }
49844|
49845| void MyInitializePagedLookasideList(
49846|   IN PPAGED_LOOKASIDE_LIST Lookaside,
49847|   IN PALLOCATE_FUNCTION Allocate,
49848|   IN PFREE_FUNCTION Free,
49849|   IN ULONG Flags,
49850|   IN ULONG Size,
49851|   IN ULONG Tag,
49852|   IN USHORT Depth
49853| )
49854| {
49855|   ASSERT(Size >= sizeof(PVOID));
49856|
         | RtlZeroMemory(Lookaside,sizeof(PAGED_LOOKASIDE_LIST));
49857|

```

```

    | InitGeneralLookaside(&Lookaside->L,Allocate,Free,Flags,S
    | ize,Tag,Depth,PagedPool);
49858|
49859|   ExInitializeFastMutex(&Lookaside->Lock);
49860|
49861|   // populate the list
49862|   // fixfixfix do we want the maximum allocated or
    | some
49863|   // percentage?
49864|
49865|   // Do this in two phases: first allocate 'Depth'
    | nodes and put into
49866|   // linked list.
49867|   PVOID AllocList=0;
49868|   PVOID Ptr=0;
49869|   ULONG NumAllocated=0;
49870|   for(USHORT i=0;i<Depth;i++) {
49871|       Ptr =
    | ExAllocateFromPagedLookasideList(Lookaside);
49872|       if(Ptr) {
49873|           ++NumAllocated;
49874|           *((void**)Ptr) = AllocList;
49875|           AllocList = Ptr;
49876|       } else {
49877|           KdPrint(("NewMem: Init: Error! out of
    | memory for list\n"));
49878|           break;
49879|       }
49880|   }
49881|
49882|   // Second phase: remove nodes from linked list one
    | by one and insert into lookaside list.
49883|   USHORT NumInserted = 0;
49884|   while (AllocList) {
49885|       Ptr = AllocList;
49886|       AllocList = *((void**)AllocList);
49887|       ++NumInserted;
49888|       ASSERT(NumInserted<=Depth);
49889|       ExFreeToPagedLookasideList(Lookaside,Ptr);
49890|   }
49891|
49892|   ASSERT(NumInserted==NumAllocated);
49893| }
49894|
49895| void MyInitializeNPagedLookasideList(
49896|     IN PNPAGED_LOOKASIDE_LIST Lookaside,
49897|     IN PALLOCATE_FUNCTION Allocate,
49898|     IN PFREE_FUNCTION Free,
49899|     IN ULONG Flags,
49900|     IN ULONG Size,

```

```

49901|    IN ULONG Tag,
49902|    IN USHORT Depth
49903|    )
49904| {
49905|
49906|     | RtlZeroMemory(Lookaside,sizeof(NPAGED_LOOKASIDE_LIST));
49907|
49908|     | InitGeneralLookaside(&Lookaside->L,Allocate,Free,Flags,S
49909|     | ize,Tag,Depth,NonPagedPool);
49910|
49911|     | KeInitializeSpinLock(&Lookaside->Lock);
49912|
49913|     // populate the list
49914|     // fixfixfix do we want the maximum allocated or
49915|     | some
49916|     // percentage?
49917|
49918|     // Do this in two phases: first allocate 'Depth'
49919|     | nodes and put into
49920|     // linked list.
49921|     PVOID AllocList=0;
49922|     PVOID Ptr=0;
49923|     ULONG NumAllocated=0;
49924|     for(USHORT i=0;i<Depth;i++) {
49925|         Ptr =
49926|         | ExAllocateFromNPagedLookasideList(Lookaside);
49927|         if(Ptr) {
49928|             ++NumAllocated;
49929|             *((void**)Ptr) = AllocList;
49930|             AllocList = Ptr;
49931|         } else {
49932|             KdPrint(("NewMem: Init: Error! out of
49933|             | memory for list\n"));
49934|             break;
49935|         }
49936|     }
49937|
49938|     // Second phase: remove nodes from linked list one
49939|     | by one and insert into lookaside list.
49940|     USHORT NumInserted = 0;
49941|     while (AllocList) {
49942|         Ptr = AllocList;
49943|         AllocList = *((void**)AllocList);
49944|         ++NumInserted;
49945|         ASSERT(NumInserted<=Depth);
49946|         ExFreeToNPagedLookasideList(Lookaside,Ptr);
49947|     }
49948|
49949|     ASSERT(NumInserted==NumAllocated);
49950| }

```

```

49943|
49944| // to determine if the look aside list needs to be
    | increased/trimmed
49945| void AdjustLookasideListDepth( PGENERAL_LOOKASIDE
    | Lookaside )
49946| {
49947| }
49948|
49949|
49950| void MemTrackInit( PUNICODE_STRING RegistryPath )
49951| {
49952|     ULONG Size = sizeof(tMemHead);
49953|     ULONG SnapShotDepth; // number of concurrent
    | snapshos
49954|     ULONG RequestDepth; // number of concurrent
    | requests (read and write)
49955|     ULONG ReadOnlyDepth; // number of concurrent open
    | files on virtual volumes
49956|     ULONG VolumeDepth; // number of concurrent volumes
    | that will be snapped
49957|     RTL_QUERY_REGISTRY_TABLE paramTable[8]={0};
49958|
49959|     // this routine is called BEFORE the registry has
    | been read
49960|     // and before we determine if debug mode is enabled
49961|     // if you need to print something to the debug port
49962|     // use KdPrint, but keep in mind that it will not
    | go to
49963|     // the debug log file.
49964|
49965|     if(MmIsThisAnNtAsSystem()) {
49966|         // server
49967|         SnapShotDepth=0x20;
49968|         RequestDepth=0x200;
49969|         ReadOnlyDepth=0x20;
49970|         VolumeDepth=0xa;
49971|     } else {
49972|         // not server, use less memory
49973|         SnapShotDepth=0xa;
49974|         VolumeDepth = 0x2;
49975|         ReadOnlyDepth = 0x4;
49976|         RequestDepth=0x20;
49977|     }
49978|
49979|     RtlZeroMemory( &paramTable[0], sizeof(paramTable)
    | );
49980|
49981|     paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
49982|     paramTable[0].Name = L"SnapShotDepth";
49983|     paramTable[0].EntryContext = &SnapShotDepth;

```



```

49984| paramTable[1].Flags = RTL_QUERY_REGISTRY_DIRECT;
49985| paramTable[1].Name = L"RequestDepth";
49986| paramTable[1].EntryContext = &RequestDepth;
49987| paramTable[2].Flags = RTL_QUERY_REGISTRY_DIRECT;
49988| paramTable[2].Name = L"ReadOnlyDepth";
49989| paramTable[2].EntryContext = &ReadOnlyDepth;
49990| paramTable[3].Flags = RTL_QUERY_REGISTRY_DIRECT;
49991| paramTable[3].Name = L"VolumeDepth";
49992| paramTable[3].EntryContext = &VolumeDepth;
49993|
49994| RtlQueryRegistryValues(
49995|     RTL_REGISTRY_ABSOLUTE |
49996|     | RTL_REGISTRY_OPTIONAL,
49997|     RegistryPath->Buffer,
49998|     &paramTable[0],
49999|     NULL,
50000|     NULL
50001| );
50002|
50003| /*
50004| Zone routines are useful for allocating a large
50005| | number of fixed size
50006| small blocks. Look-aside lists are useful for
50007| | allocating larger blocks
50008| where there is a substantial difference between the
50009| | high and low
50010| watermarks.
50011| */
50012| /*
50013| We use our own lookaside lists for the following
50014| | reasons:
50015| 1. NT ignores the maximum depth parameter
50016| 2. Its algorithm for increasing/decreasing the
50017| | depth works by
50018| looking the ratio of alloc misses to alloc
50019| | requests since the
50020| last time the function was called (every 2
50021| | seconds). If less
50022| than 10%, the depth is decreased by 10 (but not
50023| | below 4).
50024| if larger, then it is increased by
50025| | (Ratio/2)*max_depth
50026| 3. We want to keep some extra on the list for low
50027| | memory
50028| usage (though we could have a separate list)
50029| 4. I was seeing a ___LOT___ of calls to Alloc/Free
50030|
50031| A better usage would be to look at the amount of

```

```

    | times an
50023|   Alloc/Free pair could have been avoided in the last
    | X seconds.
50024|
50025|   For now, we will hardcode the values, but later on
    | we could have a
50026|   timer event executing every 2 seconds to
    | increase/trim the depth.
50027| */
50028|
50029|
    | MyInitializePagedLookasideList(&InternalSnapShot,DEBUG_A
    | LLOC,NULL,0,sizeof(pd:tInternalSnapShot)+Size,PSM_INTER
    | NAL_SNAPSHOT,(USHORT)SnapShotDepth);
50030|
    | MyInitializeNPagedLookasideList(&MasterSnapShot,DEBUG_AL
    | LOC,NULL,0,sizeof(tkSnapShotMaster)+Size,PSM_MASTER_SNAP
    | SHOT,(USHORT)SnapShotDepth);
50031|   ASSERT((SnapShotDepth*VolumeDepth)<0x10000); //
    | make sure less than a USHORT
50032|
    | MyInitializeNPagedLookasideList(&SnapShotEntry,DEBUG_ALL
    | OC,NULL,0,sizeof(tkSnapShotEntry)+Size,PSM_SNAPSHOT_ENTR
    | Y,(USHORT)SnapShotDepth*(USHORT)VolumeDepth);
50033|
    | MyInitializePagedLookasideList(&Node,DEBUG_ALLOC,NULL,0,
    | sizeof(tTreeLeaf)+Size,NODETAG,0x100);
50034|
    | MyInitializeNPagedLookasideList(&ReadOnly,DEBUG_ALLOC,NU
    | LL,0,sizeof(tOpenSnapShotFiles)+Size,PSM_READONLY_FILE_T
    | AG,(USHORT)ReadOnlyDepth);
50035|
    | MyInitializeNPagedLookasideList(&Shared,DEBUG_ALLOC,NULL
    | ,0,sizeof(pd:tShared)+Size,PSM_DICT_SHARED_TAG,(USHORT)
    | VolumeDepth);
50036|
    | ASSERT(sizeof(tWriteRequest)==sizeof(tReadRequest));
50037|
    | MyInitializeNPagedLookasideList(&Requests,DEBUG_ALLOC,NU
    | LL,0,sizeof(tWriteRequest)+Size,WRITEREQUESTTAG,(USHORT)
    | RequestDepth);
50038|
50039|   return;
50040| }
50041|
50042| void MemTrackDelnit()
50043| {
50044|   // never called
50045|   ASSERT(FALSE);
50046| }

```

```

50047|
50048| void *MemAllocatePoolWithTag( POOL_TYPE PoolType, ULONG
    | Size, ULONG Tag )
50049| {
50050| #if 0
50051|     PVOID Parent=0, GrandParent=0;
50052|     RtlGetCallersAddress (&Parent, &GrandParent);
50053|     Debug(DEBUG_MEMORY,("NEWMEM: Alloc: p=%08x gp=%08x
    | - pool=%08x, size=%08x, Tag=%08x\n",
50054|         Parent,GrandParent,
50055|         PoolType,Size,Tag
50056|     ));
50057| #endif
50058|
50059|     WORD AllocSize;
50060|     pMemHead Ptr;
50061| #define TYPE_LOOKASIDE 0
50062| #define TYPE_DIRECT    1
50063|     ULONG Type=TYPE_LOOKASIDE;
50064|     WORD HeaderSize=sizeof(tMemHead);
50065|     NTSTATUS Status=0;
50066|     ULONG MySize;
50067|
50068|     switch(Tag) {
50069|         // memory that vdisk read/write use to double
    | buffer
50070|         case PSM_VDISK_BUFFER_TAG : // 'PSvb' 1
50071|             // memory to hold original data
50072|             case BUFFTAG           : // 'PSbu' 5
50073| #if USE_PAGED_FOR_BUFF
50074|             ASSERT(PoolType == PagedPoolCacheAligned);
50075|             HeaderSize = PAGE_SIZE;
50076|             Type = TYPE_DIRECT;
50077|             Ptr = (pMemHead)ExAllocatePoolWithTag(
    | PoolType, Size+HeaderSize, Tag );
50078|             // move pointer to be under aligned page
50079|             Ptr =
    | (pMemHead)(((PCHAR)Ptr)+HeaderSize-sizeof(tMemHead));
50080| #else
50081|             HeaderSize = PAGE_SIZE;
50082|             Type = TYPE_DIRECT;
50083|             MySize = Size+HeaderSize;
50084|             Ptr = NULL;
50085|
50086|             ASSERT(GlobalSystemProcessId ==
    | PsGetCurrentProcess());
50087|             Status = ZwAllocateVirtualMemory(
50088|                 NtCurrentProcess(), // IN HANDLE
    | ProcessHandle,
50089|                 (void**)&Ptr, // IN OUT PVOID

```

```

    | *BaseAddress,
50090|         0, //    IN ULONG ZeroBits,
50091|         &MySize, //    IN OUT PSIZE_T
    | RegionSize,
50092|         MEM_RESERVE | MEM_COMMIT, //    IN
    | ULONG AllocationType,
50093|         PAGE_READWRITE//    IN ULONG Protect
50094|     );
50095|     //Debug(DEBUG_MEMORY,("NewMem:
    | AllocVirtual=%08x\n",Status));
50096|     if(NT_SUCCESS(Status)) {
50097|         ASSERT(Ptr);
50098|         ASSERT(MySize>=Size);
50099|         HeaderSize = (WORD)(MySize-Size);
50100|         Ptr =
    | (pMemHead)(((PCHAR)Ptr)+HeaderSize-sizeof(tMemHead));
50101|     } else {
50102|         ASSERT(!Ptr);
50103|     }
50104| #endif
50105|     break;
50106|
50107|     // memory holding snapshots
50108|     case PSM_INTERNAL_SNAPSHOT : // 'PSis' 1
50109|         Ptr =
    | (pMemHead)ExAllocateFromPagedLookasideList(&InternalSnap
    | Shot);
50110|         break;
50111|
50112|     // memory holding masters
50113|     case PSM_MASTER_SNAPSHOT : // 'PSma' 0
50114|         Ptr =
    | (pMemHead)ExAllocateFromNPagedLookasideList(&MasterSnapS
    | hot);
50115|         break;
50116|
50117|     // snapshot entrys
50118|     case PSM_SNAPSHOT_ENTRY : // 'PSss' 0
50119|         Ptr =
    | (pMemHead)ExAllocateFromNPagedLookasideList(&SnapShotEnt
    | ry);
50120|         break;
50121|
50122|     // tTreeLeafs
50123|     case NODETAG : // 'PSno' 1
50124|         Ptr =
    | (pMemHead)ExAllocateFromPagedLookasideList(&Node);
50125|         break;
50126|
50127|     // files that are marked as readonly

```

```

50128|     case PSM_READONLY_FILE_TAG : // 'PSro' 0
50129|         Ptr =
50130|         | (pMemHead)ExAllocateFromNPagedLookasideList(&ReadOnly);
50131|         break;
50132|         // requests coming in
50133|     case READREQUESTTAG : // 'PSrr' 0
50134|     case WRITEREQUESTTAG : // 'PSwr' 0
50135|         Ptr =
50136|         | (pMemHead)ExAllocateFromNPagedLookasideList(&Requests);
50137|         break;
50138|     case DEBUG_ENTRY_TAG:
50139|     case FILENAMETAG:
50140|         // dont want prints on this debug stuff
50141|         Ptr = (pMemHead)ExAllocatePoolWithTag(
50142|         | PoolType, Size+sizeof(tMemHead), Tag );
50143|         Type = TYPE_DIRECT;
50144|         break;
50145|         // memory to access registry
50146|     case REGISTRYTAG : // 'PSrt' 1
50147|
50148|         // memory for shared volumes
50149|     case PSM_DICT_SHARED_TAG : // 'PSsh' 0
50150|
50151|         // memory required for strings
50152|     case STRINGTAG : // 'PSst' 1
50153|
50154|         // temp memory
50155|     case THREADTAG : // 'PSth' 1
50156|
50157|         // memory holding the free space bitmap
50158|     case PSM_FREE_SPACE_TAG : // 'PSfs' 1
50159|
50160|         // memory holding headers
50161|     case PSM_DICT_HEADER_TAG : // 'PShe' 1
50162|
50163|         // memory in release build that holds a Block
50164|         | of memory so
50165|         // we can suballocate it (see mem.cpp), this is
50166|         | subdivided
50167|         // into tTreeLeaf's
50168|     case NODEMEMTAG : // 'PSnm' 1
50169|
50170|     default:
50171| #ifdef DEBUG_MEM
50172|         {
50173|             PVOID Parent=0, GrandParent=0;
50174|             RtlGetCallersAddress (&Parent,

```

```

    | &GrandParent);
50173|         Debug(DEBUG_MEMORY,("NEWMEM: Alloc : p=%08x
    | gp=%08x - pool=%08x, size=%08x, Tag=%08x\n",
50174|         Parent,GrandParent,
50175|         PoolType,Size+sizeof(tMemHead),Tag));
50176|     }
50177| #endif
50178|     Ptr = (pMemHead)ExAllocatePoolWithTag(
    | PoolType, Size+sizeof(tMemHead), Tag );
50179|     Type = TYPE_DIRECT;
50180| }
50181|
50182|
50183| if(Ptr) {
50184|     // os had memory to give us
50185|     Ptr->Type = (BYTE)Type;
50186|     Ptr->Tag = Tag;
50187|     Ptr->HeaderSize = HeaderSize;
50188|     Ptr++;
50189| }
50190|
50191| // Suppress the assert because now at least one caller
    | will handle a fail by re-asking for less
50192| // ASSERT(Ptr);
50193|
50194|     return Ptr;
50195| }
50196|
50197| VOID MemFreePool( PVOID Buffer )
50198| {
50199|     ASSERT(Buffer);
50200|     pMemHead Ptr=(pMemHead)Buffer;
50201|     Ptr--;
50202|
50203|     PVOID RealPtr=((PCHAR)(Ptr+1))-Ptr->HeaderSize;
50204|     NTSTATUS Status;
50205|     ULONG Size;
50206|
50207|     switch(Ptr->Tag) {
50208|         // memory that vdisk read/write use to double
        | buffer
50209|         case PSM_VDISK_BUFFER_TAG : // 'PSvb' 1
50210|             // memory to hold original data
50211|             case BUFFTAG : // 'PSbu' 5
50212|                 // os supplied buffer
50213|             #if USE_PAGED_FOR_BUFF
50214|                 ExFreePool(RealPtr);
50215|             #else
50216|                 Size=0;
50217|                 ASSERT(GlobalSystemProcessId ==

```

```

    | PsGetCurrentProcess());
50218|         Status = ZwFreeVirtualMemory(
50219|             NtCurrentProcess(),//    IN HANDLE
    | ProcessHandle,
50220|             &RealPtr, //    IN OUT PVOID
    | *BaseAddress,
50221|             &Size, //    IN OUT PSIZE_T RegionSize,
50222|             MEM_RELEASE //    IN ULONG FreeType
50223|         );
50224|         if(INT_SUCCESS(Status)) {
50225|             Debug(DEBUG_MEMORY,("NewMem: Error %08x
    | freeing virtual memory\n",Status));
50226|         }
50227|
50228| #endif
50229|         break;
50230|
50231|         // memory holding snapshots
50232|         case PSM_INTERNAL_SNAPSHOT : // 'PSis' 1
50233|
    | ExFreeToPagedLookasideList(&InternalSnapShot,RealPtr);
50234|         break;
50235|
50236|         // memory holding masters
50237|         case PSM_MASTER_SNAPSHOT : // 'PSma' 0
50238|
    | ExFreeToNPagedLookasideList(&MasterSnapShot,RealPtr);
50239|         break;
50240|
50241|         // snapshot entrys
50242|         case PSM_SNAPSHOT_ENTRY : // 'PSss' 0
50243|
    | ExFreeToNPagedLookasideList(&SnapShotEntry,RealPtr);
50244|         break;
50245|
50246|         // tTreeLeafs
50247|         case NODETAG : // 'PSno' 1
50248|             ExFreeToPagedLookasideList(&Node,RealPtr);
50249|             break;
50250|
50251|         // files that are marked as readonly
50252|         case PSM_READONLY_FILE_TAG : // 'PSro' 0
50253|
    | ExFreeToNPagedLookasideList(&ReadOnly,RealPtr);
50254|         break;
50255|
50256|         // requests coming in
50257|         case READREQUESTTAG : // 'PSrr' 0
50258|         case WRITEREQUESTTAG : // 'PSwr' 0
50259|

```

```

    | ExFreeToNPagedLookasideList(&Requests,RealPtr);
50260|         break;
50261|
50262|         // memory to access registry
50263|         case REGISTRYTAG           : // 'PSrt' 1
50264|
50265|         // memory for shared volumes
50266|         case PSM_DICT_SHARED_TAG   : // 'PSsh' 0
50267|
50268|         // memory required for strings
50269|         case STRINGTAG             : // 'PSst' 1
50270|
50271|         // temp memory
50272|         case THREADTAG             : // 'PSth' 1
50273|
50274|         // memory holding the free space bitmap
50275|         case PSM_FREE_SPACE_TAG    : // 'PSfs' 1
50276|
50277|         // memory holding headers
50278|         case PSM_DICT_HEADER_TAG   : // 'PShe' 1
50279|
50280|         // memory in release build that holds a Block
    | of memory so
50281|         // we can suballocate it (see mem.cpp), this is
    | subdivided
50282|         // into tTreeLeaf's
50283|         case NODEMEMTAG            : // 'PSnm' 1
50284|
50285|         default:
50286|         // os supplied buffer
50287|         ExFreePool((((PCHAR)(Ptr+1))-Ptr->HeaderSize);
50288|     }
50289| }
50290|
50291|
50292|
50293| File Listing: MEMTRACK.h
50294|
50295| PVOID MemAllocatePoolWithTag( IN POOL_TYPE PoolType, IN
    | ULONG NumberOfBytes, IN ULONG Tag );
50296| VOID MemFreePool( PVOID Buffer );
50297| #define MemAllocatePool(t,s)
    | MemAllocatePoolWithTag(t,s,DEFAULT_MEM_TAG)
50298|
50299| #define DEFAULT_MEM_TAG 0x50534d4d
50300|
50301| #define MemCheckPool(x)          1
50302| #define MemTrackPrintStats(x)
50303| #define MemShowUsage()
50304|

```



```

50305| void MemTrackInit( PUNICODE_STRING Registry );
50306|
50307| #define MemAllocateString(n)
    | (WCHAR*)MemAllocatePoolWithTag(PagedPool,(n)*sizeof(WCHA
    | R),STRINGTAG)
50308| #define MemAllocateStringByBytes(b)
    | (WCHAR*)MemAllocatePoolWithTag(PagedPool,(b),STRINGTAG)
50309| #define MemFreeString MemFreePool
50310|
50311|
50312|
50313| File Listing: MISC.cpp
50314|
50315| #include "precomp.h"
50316|
50317| /*
50318|  Kernel mode APC's can NOT be disabled (Fast mutexes
    | disable them) as the
50319|  IO manager uses them to deliver events
50320|
50321| */
50322|
50323| /*-----
    | -----*/
50324| NTSTATUS AcquireOpenCloseResource( void )
50325| {
50326|  PVOID ObjectTable[2] = { &PSMOpenCloseSemaphore,
    | &PSManExitingEvent };
50327|
50328|  ASSERT(KernelModeCurrentIrql() == PASSIVE_LEVEL);
50329|  return pmWaitForMultipleObjects( ObjectTable,2,NULL
    | );
50330| }
50331|
50332| NTSTATUS AcquireOpenCloseResourceOnly( PKEVENT
    | AbortEvent )
50333| {
50334|  PVOID ObjectTable[2] = { &PSMOpenCloseSemaphore,
    | AbortEvent};
50335|
50336|  ASSERT(KernelModeCurrentIrql() == PASSIVE_LEVEL);
50337|  return
    | pmWaitForMultipleObjects(ObjectTable,AbortEvent ? 2 :
    | 1,NULL);
50338|
50339| }
50340|
50341| /*-----
    | -----*/
50342| void ReleaseOpenCloseResource( void )

```

```

50343| {
50344|     ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
50345|     pmSignalSemaphore( &PSMOpenCloseSemaphore);
50346|
50347| }
50348|
50349| /*-----
| -----*/
50350| NTSTATUS AcquireVDiskResource( void )
50351| {
50352| //     ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
50353|     return pmAcquireSemaphore(
| &PSMVDiskSemaphore,NULL);
50354| }
50355|
50356| /*-----
| -----*/
50357| void ReleaseVDiskResource( void )
50358| {
50359| //     ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
50360|     pmSignalSemaphore( &PSMVDiskSemaphore);
50361|
50362| }
50363|
50364| #ifdef DEBUG_SNAPSHOTS
50365| NTSTATUS AcquireSnapShotResource( void )
50366| {
50367| #if 0
50368|     ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
50369|     if(pmExamineSemaphore(&PSMSnapShotSemaphore)<1) {
50370|         Debug(DEBUG_MISC,("SnapShot already Acquired!
| %d\n",pmExamineSemaphore(&PSMSnapShotSemaphore)));
50371|         DbgBreakPoint();
50372|     }
50373| #endif
50374| #ifdef DEBUG
50375|     if(IsGlobalDeviceAcquired()) {
50376|         Debug(DEBUG_MISC,("AcquireSnapShotResource:
| ERROR!!!! Global resource acquired before snapshot!
| this WILL cause a deadlock!\n"));
50377|         DbgBreakPoint();
50378|     }
50379|
50380| #endif
50381|     return pmAcquireSemaphore(
| &PSMSnapShotSemaphore,NULL);
50382| }
50383|
50384| /*-----
| -----*/

```

```

50385| void ReleaseSnapShotResource( void )
50386| {
50387| #if 0
50388|     ASSERT(KeGetCurrentIrql() == PASSIVE_LEVEL);
50389|     if(pmExamineSemaphore(&PSMSnapShotSemaphore)>0) {
50390|         Debug(DEBUG_MISC,("SnapShot not acquired!
| %d\n",pmExamineSemaphore(&PSMSnapShotSemaphore)));
50391|         DbgBreakPoint();
50392|     }
50393| #endif
50394|     pmSignalSemaphore( &PSMSnapShotSemaphore);
50395|
50396| }
50397| #endif
50398|
50399| #ifdef DEBUG
50400| /*-----
| -----*/
50401| void DumpBootSector( char *Buffer )
50402| {
50403|
50404|     if
| (strncmp(((PNTFS_BOOT_SECTOR)Buffer)->OemId,"NTFS",4)==0
| ) {
50405|         PNTFS_BOOT_SECTOR
| Ntfs=(PNTFS_BOOT_SECTOR)Buffer;
50406|
50407|         Debug(DEBUG_INFO,("NTFS Boot Sector\n"));
50408|         Debug(DEBUG_INFO,(" BytesPerSector
| %04x\n",Ntfs->BytesPerSector));
50409|         Debug(DEBUG_INFO,(" SectorsPerCluster
| %02x\n",Ntfs->SectorsPerCluster));
50410|         Debug(DEBUG_INFO,(" MediaID
| %02x\n",Ntfs->MediaID));
50411|         Debug(DEBUG_INFO,(" SectorsPerTrack
| %04x\n",Ntfs->SectorsPerTrack));
50412|         Debug(DEBUG_INFO,(" Heads
| %04x\n",Ntfs->Heads));
50413|         Debug(DEBUG_INFO,(" HiddenSectors
| %08x\n",Ntfs->HiddenSectors));
50414|         Debug(DEBUG_INFO,(" LargeSectors
| %08x\n",Ntfs->LargeSectors));
50415|         Debug(DEBUG_INFO,(" Drive
| %02x\n",Ntfs->Drive));
50416|         Debug(DEBUG_INFO,(" DirtyVolume
| %02x\n",Ntfs->DirtyVolume));
50417|         Debug(DEBUG_INFO,(" Reserved
| %04x\n",Ntfs->Reserved));
50418|         Debug(DEBUG_INFO,(" TotalSectors
| %016l64x\n",Ntfs->TotalSectors));

```

```

50419|     Debug(DEBUG_INFO,(" ClusterToMFT
| %016l64x\n",Ntfs->ClusterToMFT));
50420|     Debug(DEBUG_INFO,(" ClustersToMFTMirror
| %016l64x\n",Ntfs->ClustersToMFTMirror));
50421|     Debug(DEBUG_INFO,(" ClustersPerFRS
| %08x\n",Ntfs->ClustersPerFRS));
50422|     Debug(DEBUG_INFO,(" ClustersPerIndexBlock
| %08x\n",Ntfs->ClustersPerIndexBlock));
50423|     Debug(DEBUG_INFO,(" SerialNumber
| %016l64x\n",Ntfs->SerialNumber));
50424|     Debug(DEBUG_INFO,(" SectorChecksum
| %08x\n",Ntfs->SectorChecksum));
50425|     Debug(DEBUG_INFO,(" Unknown1
| %08x\n",Ntfs->Unknown1));
50426|     Debug(DEBUG_INFO,(" Unknown2
| %08x\n",Ntfs->Unknown2));
50427|     Debug(DEBUG_INFO,(" Unknown3
| %02x\n",Ntfs->Unknown3));
50428| } else
50429| if
| (strncmp(((PFAT_BOOT_SECTOR)Buffer)->FileSysType,"FAT",3
| )==0) {
50430|     PFAT_BOOT_SECTOR Fat=(PFAT_BOOT_SECTOR)Buffer;
50431|
50432|     Debug(DEBUG_INFO,("Fat Boot Sector\n"));
50433|     Debug(DEBUG_INFO,(" BytesPerSector
| %04x\n",Fat->BytesPerSector));
50434|     Debug(DEBUG_INFO,(" SectorsPerCluster
| %02x\n",Fat->SectorsPerCluster));
50435|     Debug(DEBUG_INFO,(" ReservedSectors
| %04x\n",Fat->ReservedSectors));
50436|     Debug(DEBUG_INFO,(" NumberOfFats
| %02x\n",Fat->NumberOfFats));
50437|     Debug(DEBUG_INFO,(" RootDirectory
| %04x\n",Fat->RootDirectory));
50438|     Debug(DEBUG_INFO,(" SmallSectors
| %04x\n",Fat->SmallSectors));
50439|     Debug(DEBUG_INFO,(" MediaID
| %02x\n",Fat->MediaID));
50440|     Debug(DEBUG_INFO,(" SectorsPerFat
| %04x\n",Fat->SectorsPerFat));
50441|     Debug(DEBUG_INFO,(" SectorsPerTrack
| %04x\n",Fat->SectorsPerTrack));
50442|     Debug(DEBUG_INFO,(" Heads
| %04x\n",Fat->Heads));
50443|     Debug(DEBUG_INFO,(" HiddenSectors
| %08x\n",Fat->HiddenSectors));
50444|     Debug(DEBUG_INFO,(" LargeSectors
| %08x\n",Fat->LargeSectors));
50445|     Debug(DEBUG_INFO,(" Drive

```

```

    | %02x\n",Fat->Drive));
50446|    Debug(DEBUG_INFO,(" DirtyVolume
    | %02x\n",Fat->DirtyVolume));
50447|    Debug(DEBUG_INFO,(" BootSignature
    | %02x\n",Fat->BootSignature));
50448|    Debug(DEBUG_INFO,(" VolumeID
    | %08x\n",Fat->VolumeID));
50449|    Debug(DEBUG_INFO,(" VolumeLabel
    | %-11.11s\n",Fat->VolumeLabel));
50450|    Debug(DEBUG_INFO,(" FileSysType
    | %-8.8s\n",Fat->FileSysType));
50451|    } else
50452|    if
    | (strncmp(((PFAT32_BOOT_SECTOR)Buffer)->FileSysType,"FAT3
    | 2",3)==0) {
50453|    PFAT32_BOOT_SECTOR
    | Fat=(PFAT32_BOOT_SECTOR)Buffer;
50454|
50455|    Debug(DEBUG_INFO,("Fat32 Boot Sector\n"));
50456|    Debug(DEBUG_INFO,(" BytesPerSector
    | %04x\n",Fat->BytesPerSector));
50457|    Debug(DEBUG_INFO,(" SectorsPerCluster
    | %02x\n",Fat->SectorsPerCluster));
50458|    Debug(DEBUG_INFO,(" ReservedSectors
    | %04x\n",Fat->ReservedSectors));
50459|    Debug(DEBUG_INFO,(" NumberOfFats
    | %02x\n",Fat->NumberOfFats));
50460|    Debug(DEBUG_INFO,(" RootDirectory
    | %04x\n",Fat->RootEntries));
50461|    Debug(DEBUG_INFO,(" SmallSectors
    | %04x\n",Fat->SmallSectors));
50462|    Debug(DEBUG_INFO,(" MediaID
    | %02x\n",Fat->MediaID));
50463|    Debug(DEBUG_INFO,(" SectorsPerFat
    | %04x\n",Fat->SectorsPerFat));
50464|    Debug(DEBUG_INFO,(" SectorsPerTrack
    | %04x\n",Fat->SectorsPerTrack));
50465|    Debug(DEBUG_INFO,(" Heads
    | %04x\n",Fat->Heads));
50466|    Debug(DEBUG_INFO,(" HiddenSectors
    | %08x\n",Fat->HiddenSectors));
50467|    Debug(DEBUG_INFO,(" LargeSectors
    | %08x\n",Fat->LargeSectors));
50468|
50469|    Debug(DEBUG_INFO,(" LargeSectors Fat
    | %08x\n",Fat->LargeSectorsPerFat));
50470|    Debug(DEBUG_INFO,(" ExtendedFlags
    | %04x\n",Fat->ExtendedFlags));
50471|    Debug(DEBUG_INFO,(" FsVersion
    | %04x\n",Fat->FsVersion));

```

```

50472|     Debug(DEBUG_INFO,(" RootDirCluster
    | %08x\n",Fat->RootDirFirstCluster));
50473|     Debug(DEBUG_INFO,(" FsInfoSector
    | %04x\n",Fat->FsInfoSector));
50474|     Debug(DEBUG_INFO,(" BackupBootSector
    | %04x\n",Fat->BackupBootSector));
50475|
50476|     Debug(DEBUG_INFO,(" Drive
    | %02x\n",Fat->Drive));
50477|     Debug(DEBUG_INFO,(" DirtyVolume
    | %02x\n",Fat->DirtyVolume));
50478|     Debug(DEBUG_INFO,(" BootSignature
    | %02x\n",Fat->BootSignature));
50479|     Debug(DEBUG_INFO,(" VolumeID
    | %08x\n",Fat->VolumeID));
50480|     Debug(DEBUG_INFO,(" VolumeLabel
    | %-11.11s\n",Fat->VolumeLabel));
50481|     Debug(DEBUG_INFO,(" FileSysType
    | %-8.8s\n",Fat->FileSysType));
50482| } else {
50483|     Debug(DEBUG_INFO,("Unknown boot sector\n"));
50484|     DumpSector(Buffer,512);
50485| }
50486|
50487|
50488| }
50489|
50490| /*-----
    | -----*/
50491| void DumpSector( char *Buffer, ULONG Size )
50492| {
50493|     ULONG i,j;
50494|     UCHAR *UBuffer = (UCHAR*)Buffer;
50495|     char s[80];
50496|     char *t;
50497|
50498| // 00: 0000: 00 00 00 00 00 00 00 00 00 00 00 00 00
    | 00 00 .....
50499|
50500| // no need to even try if debugging is not on..
50501| if(!DebugLevel) {
50502|     return;
50503| }
50504|
50505| for(i=0;i<Size;i+=16) {
50506|
50507|     t=s;
50508|
50509|     t+=sprintf(t,"%02x: %04x : ",i / 512,i);
50510|

```

```

50511|     for(j=i;j-i<16;j++) {
50512|         if (j<Size) {
50513|             t+=sprintf(t,"%02x ",UBuffer[j]);
50514|         } else {
50515|             t+=sprintf(t," ");
50516|         }
50517|     }
50518|
50519|     for(j=i;j-i<16;j++) {
50520|         if (j>=Size) {
50521|             break;
50522|         }
50523|         if ((UBuffer[j]>31) && (UBuffer[j]<128)) {
50524|             *t++ = UBuffer[j];
50525|         } else {
50526|             *t++ = '.';
50527|         }
50528|     }
50529|     *t=0;
50530|     Debug(DEBUG_INFO,("%s\n",s));
50531| }
50532| }
50533| #endif
50534|
50535| /*-----
| -----*/
50536| USHORT NumChars( const WCHAR *string )
50537| {
50538|     const WCHAR *p=string;
50539|
50540|     while(*p != '\0') {
50541|         p++;
50542|     }
50543|     return (USHORT)(p-string);
50544| }
50545|
50546| USHORT NumBytes( const WCHAR *string )
50547| {
50548|     return NumChars(string)*sizeof(WCHAR);
50549| }
50550|
50551| // 0-15 if one, -1 if not a hex char
50552| /*-----
| -----*/
50553| int NumDigit( char ch, int Base )
50554| {
50555|     int Value=0;
50556|
50557|     if(isdigit(ch)) {
50558|         Value = ch-'0';

```

```

50559| } else
50560| if((toupper(ch)>='A') && (toupper(ch)<='Z')) {
50561|     Value = toupper(ch)-'A'+10;
50562| } else {
50563|     Value = Base+1; // error out
50564| }
50565|
50566| if(Value<=Base) {
50567|     //Debug(DEBUG_INFO,("NumDigit: ch='%c',
    | base=%d, returning %d\n",ch,Base,Value));
50568|     return Value;
50569| }
50570|
50571| //Debug(DEBUG_INFO,("NumDigit: ch='%c', base=%d,
    | returning -1\n",ch,Base));
50572| return -1;
50573| }
50574|
50575| /*-----
    | -----*/
50576|
50577| int __iswdigit( wint_t ch )
50578| {
50579|     if((ch>=L'0') && (ch<=L'9')) {
50580|         return TRUE;
50581|     } else {
50582|         return FALSE;
50583|     }
50584| }
50585|
50586| // 0-15 if one, -1 if not a hex char
50587| /*-----
    | -----*/
50588| int WNumDigit( WCHAR ch, int Base )
50589| {
50590|     int Value=0;
50591|
50592|     if(__iswdigit(ch)) {
50593|         Value = ch-L'0';
50594|     } else
50595|     if((towupper(ch)>=L'A') && (towupper(ch)<=L'Z')) {
50596|         Value = towupper(ch)-L'A'+10;
50597|     } else {
50598|         Value = Base+1; // error out
50599|     }
50600|
50601|     if(Value<=Base) {
50602|         return Value;
50603|     }
50604|

```



```

50605|    return -1;
50606| }
50607|
50608| /*-----
    | -----*/
50609| ULONG WAsciiToInt( WCHAR **Line, int Base )
50610| {
50611|     int Accum = 0;
50612|     int Digit;
50613|     int Times=1;
50614|
50615|     while(**Line) {
50616|         WCHAR ch=**Line;
50617|         if((ch==L',') || (ch==L' ') || (ch==L'\t') ||
            | (ch==L'\n') || (ch==L'\r')) {
50618|             (*Line)++;
50619|         } else {
50620|             break;
50621|         }
50622|     }
50623|
50624|     if(**Line==L'-') {
50625|         Times = -1;
50626|         (*Line)++;
50627|     }
50628|
50629|     while((Digit=WNumDigit(**Line,Base))>=0) {
50630|         (*Line)++;
50631|         Accum *= Base;
50632|         Accum += Digit;
50633|     }
50634|     return (ULONG)(Accum * Times);
50635|
50636| }
50637|
50638| WCHAR *_ltow( ULONG Value, WCHAR *String, ULONG Base,
            | ULONG StrLen )
50639| {
50640|     UNICODE_STRING Uni;
50641|     RtlInitUnicodeString(&Uni,String);
50642|     Uni.Length = 0;
50643|     Uni.MaximumLength = (USHORT)StrLen;
50644|     RtlIntegerToUnicodeString(Value,Base,&Uni);
50645|     return String;
50646| }
50647|
50648|
50649| #ifdef DEBUG
50650| #define DEBUGEXCEPTION
50651| #endif

```

```

50652|
50653| typedef struct sMyExceptionInfo {
50654|     PVOID DriverStart;
50655|     ULONG DriverSize;
50656|     EXCEPTION_RECORD ExceptionRecord;
50657| } MyExceptionInfo;
50658|
50659| DWORD ExceptionFilter( EXCEPTION_POINTERS *ep )
50660| {
50661|     MyExceptionInfo Local;
50662|
50663|     Debug(DEBUG_INFO,("Exception occurred, ep=%08x,
        | '.cxr %08x' to switch to
        | context\n",ep,ep->ContextRecord));
50664|     Debug(DEBUG_INFO,(" Exception:
        | %08x\n",ep->ExceptionRecord->ExceptionCode));
50665|     Debug(DEBUG_INFO,(" Flags :
        | %08x\n",ep->ExceptionRecord->ExceptionFlags));
50666|     Debug(DEBUG_INFO,(" Address :
        | %08x\n",ep->ExceptionRecord->ExceptionAddress));
50667|     Debug(DEBUG_INFO,(" Base :
        | %08x\n",PSManDriverObject->DriverStart));
50668|     Debug(DEBUG_INFO,(" Size :
        | %08x\n",PSManDriverObject->DriverSize));
50669|
50670|     | ASSERT(sizeof(*ep->ExceptionRecord)==sizeof(EXCEPTION_RE
        | CORD));
50671|
        | memcpy(&Local.ExceptionRecord,ep->ExceptionRecord,sizeof
        | (*ep->ExceptionRecord));
50672|     Local.DriverStart = PSManDriverObject->DriverStart;
50673|     Local.DriverSize = PSManDriverObject->DriverSize;
50674|
50675|     /*lint -save -e740 */
50676|
        | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_EXCE
        | PTION_OCCURED,0,&Local,sizeof(Local),NULL,0);
50677|     /*lint -restore */
50678|
50679|     switch ( ep->ExceptionRecord->ExceptionCode ) {
50680|
50681|         case STATUS_ACCESS_VIOLATION:
50682|             Debug(DEBUG_INFO,("The instruction at
                | \"0x%08lx\" referenced memory at \"0x%08lx\". The
                | memory\n"
50683|
                    | "could not be \"%s\".\n",
50684|
                | ep->ExceptionRecord->ExceptionAddress,
50685|

```

```

    | ep->ExceptionRecord->ExceptionInformation[1],
50686|
    | ep->ExceptionRecord->ExceptionInformation[0] ?
    | "written" : "read"
50687|     ));
50688|
50689| #ifdef DEBUGEXCEPTION
50690|     if(KdDebuggerEnabled) {
50691|         DbgBreakPoint();
50692|     }
50693| #endif
50694|     return (EXCEPTION_EXECUTE_HANDLER);
50695| case STATUS_IN_PAGE_ERROR:
50696|     Debug(DEBUG_INFO,("The instruction at
    | \"0x%08lx\" referenced memory at \"0x%08lx\". The
    | required\n"
50697|         "data was not placed into
    | memory because of an I/O error status of
    | \"0x%08lx\".\n",
50698|
    | ep->ExceptionRecord->ExceptionAddress,
50699|
    | ep->ExceptionRecord->ExceptionInformation[0],
50700|
    | ep->ExceptionRecord->ExceptionInformation[1]
50701|     ));
50702|
50703| #ifdef DEBUGEXCEPTION
50704|     if(KdDebuggerEnabled) {
50705|         DbgBreakPoint();
50706|     }
50707| #endif
50708|     return (EXCEPTION_EXECUTE_HANDLER);
50709|
50710| default:
50711| #ifdef DEBUGEXCEPTION
50712|     if(KdDebuggerEnabled) {
50713|         DbgBreakPoint();
50714|     }
50715| #endif
50716|     // pass to debugger or system
50717|     return (EXCEPTION_CONTINUE_SEARCH);
50718|     // EXCEPTION_CONTINUE_SEARCH = continue looking
    | for a error handler
50719|     // EXCEPTION_EXECUTE_HANDLER = execute
    | exception handler
50720|     // EXCEPTION_CONTINUE_EXECUTION = continue as
    | though nothing happened
50721|
50722| }

```

```

50723|
50724| }
50725|
50726| void DumpObject( PDEVICE_OBJECT DevObj )
50727| {
50728|     switch(PsmGetObjectTypes(DevObj)) {
50729|         case OBJECT_FS_OBJECT : {
50730|             Debug(DEBUG_VDISK,("%08x: FileSystem
50731| | Object\n",DevObj));
50732|             break;
50733|         }
50734|         case OBJECT_FS_FILTER : {
50735|             Debug(DEBUG_VDISK,("%08x: FileSystem
50736| | Filter\n",DevObj));
50737|             break;
50738|         }
50739|         case OBJECT_VIRTUALDISK: {
50740|             PVDISK_EXTENSION DevExt =
50741| | GetVDiskExtension(DevObj);
50742|             Debug(DEBUG_VDISK,("%08x: Virtual Disk for
50743| | '%S':%d\n",DevObj,DevExt->Name,DevExt->Instance));
50744|             Debug(DEBUG_VDISK,(" PSMDevice =
50745| | %08x\n",DevExt->PSMDevice));
50746|             Debug(DEBUG_VDISK,(" Number of reads = %d
50747| | (%d), Write = %d
50748| | (%d)\n",DevExt->NumberOfReadRequests,DevExt->SectorsRead
50749| | ,DevExt->NumberOfWriteRequests,
50750| | DevExt->SectorsWritten));
50751|             Debug(DEBUG_VDISK,(" Cyls=%l64d, Heads=%d,
50752| | SPT=%d\n",DevExt->Cylinders, DevExt->Heads, DevExt->SPT
50753| | ));
50754|             Debug(DEBUG_VDISK,(" !Ready=%d,
50755| | Active=%d\n",DevExt->DriveNotReady,DevExt->PartitionActi
50756| | ve));
50757|             Debug(DEBUG_VDISK,(" DiskChangeCount=%d,
50758| | LockCount=%d,
50759| | Keep=%d\n",DevExt->DiskChangeCount,DevExt->LockCount,Dev
50760| | Ext->KeepWriteInMemory));
50761|             Debug(DEBUG_VDISK,(" SnapShot=%08x,
50762| | Master=%08x\n",DevExt->SnapShot,DevExt->MasterSnapShot))
50763| | ;
50764|             break;
50765|         }
50766|         case OBJECT_FILTEREDDISK: {
50767|             PFILTERED_EXTENSION DevExt =
50768| | GetFilteredExtension(DevObj);
50769|             Debug(DEBUG_VDISK,("%08x: Filtered Disk for
50770| | %S\n",DevObj,DevExt->Name));
50771|             Debug(DEBUG_VDISK,("
50772| | TargetDeviceObject %08x\n"

```

```

50753|         " NumReads %d, %d, NumWrites %d, %d\n"
50754|         " ChangeCount %d, CacheWrites %d,
    | PSMed %d, SR %d, SW %d, OpenCount %d\n"
50755|         " SnapShots %08x\n",
50756|
50757|         DevExt->TargetDeviceObject,
50758|         DevExt->NumberOfReadRequests,
50759|         DevExt->SectorsRead,
50760|         DevExt->NumberOfWriteRequests,
50761|         DevExt->SectorsWritten,
50762|         DevExt->ChangeCount,
50763|         DevExt->CacheWrites,
50764|         DevExt->PSMed,
50765|         DevExt->SignalRead,
50766|         DevExt->SignalWrite,
50767|         DevExt->OpenCount,
50768|         &DevExt->SnapShots
50769|     ));
50770|
50771|     Debug(DEBUG_VDISK,(
50772|         " Logical: Start %08x%08x Length
    | %08x%08x Hidden %d, Type %d\n",
50773|         // logical devices only
50774|         DevExt->Pi.StartingOffset.HighPart,
50775|         DevExt->Pi.StartingOffset.LowPart,
50776|         DevExt->Pi.PartitionLength.HighPart,
50777|         DevExt->Pi.PartitionLength.LowPart,
50778|         DevExt->Pi.HiddenSectors,
50779|         DevExt->Pi.PartitionType
50780|     ));
50781|     Debug(DEBUG_VDISK,(
50782|         " Physical: Sig %08x, Last %d, Cyls
    | %08x%08x Heads %d, SPT %d, BPS %d\n",
50783|         // physical devices only!!!!
50784|         DevExt->Physical.Signature,
50785|         DevExt->Physical.LastPartitionNumber,
50786|         DevExt->Cylinders.HighPart,
50787|         DevExt->Cylinders.LowPart,
50788|         DevExt->TracksPerCylinder,
50789|         DevExt->SectorsPerTrack,
50790|         DevExt->BytesPerSector
50791|     ));
50792|
50793|     break;
50794| }
50795| case OBJECT_INTERNAL: {
50796|     PSBPSMAN_EXTENSION
    | DevExt=(PSBPSMAN_EXTENSION)GetDeviceExtension(DevObj);
50797|     Debug(DEBUG_VDISK,("%08x: Internal
    | Object\n"));

```

```

50798|         Debug(DEBUG_VDISK,(" Psm Users=%08x, Num
    | Active=%d\n",DevExt->PSMUsers,DevExt->NumActive));
50799|         break;
50800|     }
50801| }
50802| }
50803|
50804| void DumpAllDisks()
50805| {
50806|     PDEVICE_OBJECT DevObj =
    | PSMANDriverObject->DeviceObject;
50807|
50808|     while(DevObj!=NULL) {
50809|         DumpObject(DevObj);
50810|         DevObj=DevObj->NextDevice;
50811|     }
50812| }
50813|
50814| //-----
    | -----
50815|
50816| #ifdef DEBUG
50817| bool ProfilerResourceInitialized = false;
50818| bool ProfilerEnabled = true;
50819| ERESOURCE ProfilerResource = {0};
50820| KSPIN_LOCK ProfilerSpinLock = {0};
50821| ProfileNode *ProfileList = 0;
50822| ProfileNode *ProfileFreeList = 0;
50823| ProfileChunk *ProfileChunkList = 0;
50824| const int PROFILE_NODES_PER_CHUNK = 4000 /
    | sizeof(ProfileNode);
50825| ULONG ProfileNumNodes = 0;
50826| ULONG ProfileNumChunks = 0;
50827|
50828| void InitProfiler()
50829| {
50830|     if ( !ProfilerResourceInitialized ) {
50831|         ExInitializeResourceLite (&ProfilerResource);
50832|         KeInitializeSpinLock(&ProfilerSpinLock);
50833|         ProfilerResourceInitialized = true;
50834|     }
50835| }
50836| #endif /*DEBUG*/
50837|
50838| //-----
    | -----
50839|
50840| #ifdef DEBUG
50841| ProfileNode *AllocProfileNode()
50842| {

```

```

50843| ProfileNode *node = 0;
50844|
50845| ASSERT(ProfilerEnabled);
50846| if ( ProfilerEnabled ) {
50847|     if ( !ProfileFreeList ) {
50848|         ProfileChunk *chunk = (ProfileChunk *)
50849|         | MemAllocatePoolWithTag (PagedPool,
50850|         | sizeof(ProfileChunk), TEMPTAG);
50851|         if ( chunk ) {
50852|             chunk->array = (ProfileNode *)
50853|             | MemAllocatePoolWithTag (NonPagedPool,
50854|             | sizeof(ProfileNode)*PROFILE_NODES_PER_CHUNK, TEMPTAG);
50855|             if ( chunk->array ) {
50856|                 ProfileNode *prevNode = 0;
50857|                 for ( int i=0;
50858|                     | i<PROFILE_NODES_PER_CHUNK; ++i ) {
50859|                     chunk->array[i].next =
50860|                     | prevNode;
50861|                     prevNode = &(chunk->array[i]);
50862|                 }
50863|                 ProfileFreeList = prevNode;
50864|                 chunk->next = ProfileChunkList;
50865|                 ProfileChunkList = chunk;
50866|                 ++ProfileNumChunks;
50867|             } else {
50868|                 MemFreePool(chunk);
50869|                 chunk = 0;
50870|             }
50871|         }
50872|     }
50873| }
50874|
50875| if ( ProfileFreeList ) {
50876|     node = ProfileFreeList;
50877|     ProfileFreeList = ProfileFreeList->next;
50878|     node->next = 0;
50879|     ++ProfileNumNodes;
50880| }
50881|
50882| return node;
50883| }
50884| #endif /*DEBUG*/
50885|
50886| //-----
50887| | -----
50888|
50889| #ifdef DEBUG
50890| void ProfileFunc (
50891|     const char    *name,
50892|     PROFILE_COUNT *counter,

```

```

50886|  const char    *sourceFileName,
50887|  int          sourceLineNumber )
50888|  {
50889|  __try {
50890|      if ( ProfilerEnabled ) {
50891|          ASSERT(ProfilerResourceInitialized);
50892|          if ( ProfilerResourceInitialized ) {
50893|              if ( !counter ) {
50894|
50895|                  | MyAcquireResourceExclusiveLite(&ProfilerResource,
50896|                  | TRUE);
50897|
50898|                  __try {
50899|                      // Test 'counter' again,
50900|                      | because it may have been initialized by another thread
50901|                      | before we acquired resource
50902|                      if ( !counter ) {
50903|                          // WARNING: All strings
50904|                          | passed in must be static, hardcoded strings! No
50905|                          | stack/dynamic variables, please!
50906|                          // NOTE: Allocate from
50907|                          | non-paged pool because we will access memory while
50908|                          | holding spin lock.
50909|                          ProfileNode *node =
50910|                          | AllocProfileNode();
50911|                          if ( node ) {
50912|                              node->initialize (name,
50913|                              | sourceFileName, sourceLineNumber);
50914|                              counter =
50915|                              | &(node->counter);
50916|                              node->next =
50917|                              | ProfileList;
50918|                              ProfileList = node;
50919|                          }
50920|                      }
50921|                  } __finally {
50922|                      | MyReleaseResourceForThreadLite(&ProfilerResource);
50923|                      }
50924|                  }
50925|
50926|                  if ( counter ) {
50927|                      KIRQL oldIrql;
50928|                      pmAcquireSpinLock
50929|                      | (&ProfilerSpinLock, &oldIrql);
50930|                      ++(*counter);
50931|                      pmReleaseSpinLock
50932|                      | (&ProfilerSpinLock, oldIrql);
50933|                  }
50934|              }
50935|          }
50936|      }
50937|  }

```



```

50921|    } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
50922|        NTSTATUS Status = GetExceptionCode();
50923|        Debug(DEBUG_MISC,("!!! ProfileFunc: Exception
    | %08x\n",Status));
50924|    }
50925| }
50926| #endif /*DEBUG*/
50927|
50928| //-----
    | -----
50929|
50930| #ifdef DEBUG
50931| void DumpProfileInfo()
50932| {
50933|     __try {
50934|         if ( ProfilerEnabled &&
    | ProfilerResourceInitialized ) {
50935|
    | MyAcquireResourceExclusiveLite(&ProfilerResource,TRUE);
50936|         __try {
50937|             Debug(DEBUG_MISC,("Profiler stats:
    | nodes=%08x,
    | chunks=%08x\n",ProfileNumNodes,ProfileNumChunks));
50938|             Debug(DEBUG_MISC,("Profiler Dump Begins
    | -----\n
    | "));
50939|
50940|             // First find maximum length of any
    | counter...
50941|             int MaxCountLen = 0;
50942|             int MaxNameLen = 0;
50943|             ProfileNode *node;
50944|             for ( node=ProfileList; node;
    | node=node->next ) {
50945|                 char temp [32];
50946|
    | sprintf(temp,"%l64x",node->counter);
50947|                 int len = strlen(temp);
50948|                 if ( len > MaxCountLen ) {
50949|                     MaxCountLen = len;
50950|                 }
50951|
    | len = strlen(node->name);
50952|                 if ( len > MaxNameLen ) {
50953|                     MaxNameLen = len;
50954|                 }
50955|             }
50956|         }
50957|
    | for ( node=ProfileList; node;

```

```

    | node=node->next ) {
50959|         Debug(DEBUG_MISC,(" %*164x: %-*s |
    | %s[%d]\n",
50960|             MaxCountLen,
50961|             node->counter,
50962|             MaxNameLen,
50963|             node->name,
50964|             node->sourceFileName,
50965|             node->sourceLineNumber));
50966|     }
50967|     Debug(DEBUG_MISC,("Profiler Dump Ends
    | -----\n
    | "));
50968|     } __finally {
50969|
    | MyReleaseResourceForThreadLite(&ProfilerResource);
50970|     }
50971|     }
50972| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
50973|     NTSTATUS Status = GetExceptionCode();
50974|     Debug(DEBUG_MISC,("!!! DumpProfileInfo:
    | Exception %08x\n",Status));
50975| }
50976| }
50977| #endif /*DEBUG*/
50978|
50979| //-----
    | -----
50980|
50981| void DumpProfileInfo_Thread ( void * )
50982| {
50983|     DumpProfileInfo();
50984| }
50985|
50986| //-----
    | -----
50987| // StopWatch code starts....
50988|
50989| #ifdef DEBUG
50990|
50991| PSM_StopWatch *PSM_StopWatch::All = 0;
50992|
50993| PSM_StopWatch::PSM_StopWatch ( const char *_name ):
50994|     name(_name)
50995| {
50996|     KIRQL OldIrql = 0;
50997|
50998|     KeInitializeSpinLock (&spinLock);
50999|     KeAcquireSpinLock (&spinLock, &OldIrql);

```

```

51000|  totalTime = minTime = maxTime = 0;
51001|  callCount = 0;
51002|  nestingLevel = 0;
51003|  next = All;
51004|  All = this;
51005|  KeReleaseSpinLock (&spinLock, OldIrql);
51006| }
51007|
51008| void PSM_StopWatch::start()
51009| {
51010|  KIRQL OldIrql = 0;
51011|  KeAcquireSpinLock ( &spinLock, &OldIrql );
51012|  if ( nestingLevel++ == 0 ) {
51013|      ++callCount;
51014|      startTime = ULONGLONG
        | (KeQueryPerformanceCounter(NULL).QuadPart);
51015|  }
51016|  KeReleaseSpinLock ( &spinLock, OldIrql );
51017| }
51018|
51019| void PSM_StopWatch::pause()
51020| {
51021|  KIRQL OldIrql = 0;
51022|  KeAcquireSpinLock ( &spinLock, &OldIrql );
51023|  if ( --nestingLevel == 0 ) {
51024|      LARGE_INTEGER timerTicksPerSecond;
51025|      ULONGLONG finishTime =
        | KeQueryPerformanceCounter(&timerTicksPerSecond).QuadPart
        | ;
51026|      ULONGLONG timerTicksElapsed = finishTime -
        | startTime;
51027|
51028|      // We want the number of microseconds.
51029|      // 'timerTicksElapsed' holds how many timer
        | ticks happened between starting and stopping.
51030|      // 'timerTicksPerSecond' holds how many timer
        | ticks happen every second.
51031|      // ticks / ticks-per-second = seconds, but then
        | multiply by a million to get microseconds.
51032|
51033|      ASSERT(timerTicksPerSecond.QuadPart > 0);
51034|      ULONGLONG interval = timerTicksElapsed *
        | ULONGLONG(1000000) / timerTicksPerSecond.QuadPart;
        | // convert to microseconds
51035|
51036|      totalTime += interval;
51037|      if ( callCount==1 || interval>maxTime ) {
51038|          maxTime = interval;
51039|      }
51040|      if ( callCount==1 || interval<minTime ) {

```

```

51041|         minTime = interval;
51042|     }
51043| }
51044| KeReleaseSpinLock ( &spinLock, OldIrql );
51045| }
51046|
51047| void PSM_StopWatch::reset()
51048| {
51049|     KIRQL OldIrql = 0;
51050|     KeAcquireSpinLock ( &spinLock, &OldIrql );
51051|     if ( nestingLevel == 0 ) {
51052|         startTime = totalTime = minTime = maxTime = 0;
51053|         callCount = 0;
51054|     }
51055|     KeReleaseSpinLock ( &spinLock, OldIrql );
51056| }
51057|
51058| void PSM_StopWatch::dump()
51059| {
51060|     if ( callCount > 0 ) {
51061|         Debug(DEBUG_MISC,("StopWatch: avg=%10I64d :
| total=%10I64d : count=%10I64d : min=%10I64d :
| max=%10I64d : %s\n",
51062|             totalTime / callCount,
51063|             totalTime,
51064|             callCount,
51065|             minTime,
51066|             maxTime,
51067|             name));
51068|     } else {
51069|         Debug(DEBUG_MISC,("StopWatch: -----
| : ----- : count=      0 : -----
| : ----- : %s\n", name));
51070|     }
51071| }
51072|
51073| void PSM_StopWatch::DumpAll()
51074| {
51075|     Debug(DEBUG_MISC,("Begin StopWatch Report (all
| times in microseconds) ----\n"));
51076|     int numStopWatches = 0;
51077|     for ( PSM_StopWatch *node=All; node;
| node=node->next ) {
51078|         ++numStopWatches;
51079|         node->dump();
51080|     }
51081|
51082|     Debug(DEBUG_MISC,("End StopWatch Report (%d found)
| -----\n",numStopWatches));
51083| }

```

```

51084|
51085| void PSM_StopWatch::ResetAll()
51086| {
51087|     for ( PSM_StopWatch *node=All; node;
51088|           | node=node->next ) {
51089|         node->reset();
51090|     }
51091| }
51092| #endif /*DEBUG*/
51093|
51094| // ... StopWatch code ends
51095| //-----
51096| | -----
51097|
51098|
51099| File Listing: MISC.h
51100|
51101| inline ULONG IsValidHandle ( HANDLE Handle )
51102| {
51103|     return (Handle!=(void*)0) && (Handle!=(void*)(-1));
51104| }
51105|
51106| NTSTATUS ValidateKernelSnapShotPointer ( PVOID
51107|     | KernelPointer );
51108|
51108| void TdGetRegistrySettings ( IN PUNICODE_STRING
51109|     | RegistryPath );
51109| void *AllocBufferBelow16Meg( size_t size, ULONG Align
51110|     | );
51110| void FreeBufferBelow16Meg( void *Buffer );
51111| //NTSTATUS AcquireTapeResource( void );
51112| //void ReleaseTapeResource( void );
51113| USHORT NumChars( const WCHAR *string );
51114| USHORT NumBytes( const WCHAR *string );
51115| NTSTATUS AcquireOpenCloseResource( void );
51116| NTSTATUS AcquireOpenCloseResourceOnly( PKEVENT
51117|     | AbortEvent );
51117| void ReleaseOpenCloseResource( void );
51118| NTSTATUS AcquireVDiskResource( void );
51119| void ReleaseVDiskResource( void );
51120|
51121| #ifdef DEBUG
51122| void DumpSector( char *Buffer, ULONG Size );
51123| void DumpBootSector( char *Buffer );
51124| #else
51125| #define DumpSector(b,s)
51126| #define DumpBootSector(b)
51127| #endif

```

```

51128|
51129| DWORD ExceptionFilter( EXCEPTION_POINTERS *ep );
51130| void DumpObject( PDEVICE_OBJECT DevObj );
51131| void DumpAllDisks(void);
51132| // for testig
51133| #ifdef DEBUG_SNAPSHOTS
51134| void ReleaseSnapShotResource( void );
51135| NTSTATUS AcquireSnapShotResource( void );
51136| #endif
51137|
51138| ULONG AsciiToInt( WCHAR **Line, int Base );
51139| ULONG WAsciiToInt( WCHAR **Line, int Base );
51140| WCHAR *_ltow( ULONG Value, WCHAR *String, ULONG Base,
| ULONG StrLen );
51141| NTSTATUS GetNumActiveSnapShots ( ULONG &count );
51142|
51143| //-----
| -----
| -
51144| // This block of code implements a simple profiler.
| This is not for timing things! It is for
51145| // counting up how many times a particular line of
| code is executed.
51146| // If you want to time things, use class Stopwatch,
| below.
51147| #ifdef DEBUG
51148|     typedef __int64 PROFILE_COUNT;
51149|
51150|     struct ProfileNode {
51151|         ProfileNode    *next;           // link to
| next node in list
51152|         PROFILE_COUNT  counter;         // how many
| times this place in the code has been executed
51153|         const char    *name;           // symbolic
| name associated with the place in the code being
| profiled
51154|         const char    *sourceFileName; // name of
| source file where this entry resides
51155|         int           sourceLineNumber; // line
| number in source file where this entry resides
51156|
51157|         void initialize ( const char *_name, const char
| *_sourceFileName, int _sourceLineNumber )
51158|         {
51159|             next = 0;
51160|             counter = 0;
51161|             name = _name;
51162|             sourceFileName = _sourceFileName;
51163|             sourceLineNumber = _sourceLineNumber;
51164|         }

```

```

51165|   };
51166|
51167|   struct ProfileChunk {
51168|       ProfileNode   *array;
51169|       ProfileChunk  *next;
51170|   };
51171|
51172|   void InitProfiler();
51173|   void ProfileFunc ( const char *name, PROFILE_COUNT
    | *&counter, const char *sourceFileName, int
    | sourceLineNumber );
51174|   void DumpProfileInfo();
51175|
51176|   #define Profile(name) {static PROFILE_COUNT
    | *counter=0;
    | ProfileFunc((name),counter,__FILE__,__LINE__);}
51177| #else /*DEBUG*/
51178|   #define InitProfiler()          ((void)(0))
51179|   #define Profile(name)          ((void)(0))
51180|   #define DumpProfileInfo()      ((void)(0))
51181| #endif /*DEBUG*/
51182| //-----
    | -----
    | -
51183|
51184|
51185| //-----
    | -----
    | -
51186| //  This block of code implements class StopWatch.
    | This class is for adding up the amount of time
51187| //  that certain ranges of code use up.
51188| //-----
    | -----
    | -
51189|
51190| #ifdef DEBUG
51191|   class PSM_StopWatch
51192|   {
51193|   public:
51194|       PSM_StopWatch ( const char *_name );
51195|       // ~PSM_StopWatch() {ASSERT(FALSE);}    //
    | These objects should never be destructed!
51196|
51197|       void reset();
51198|       void start();
51199|       void pause();
51200|       void dump();
51201|
51202|       static void DumpAll();

```

```

51203|     static void ResetAll();
51204|
51205| private:
51206|     ULONGLONG     startTime;
51207|     ULONGLONG     totalTime;
51208|     ULONGLONG     minTime;
51209|     ULONGLONG     maxTime;
51210|     PSM_StopWatch *next;    // for
    | implementing list 'All'.
51211|     KSPIN_LOCK     spinLock;
51212|     int             nestingLevel;
51213|     const char * const name;
51214|     ULONGLONG     callCount;
51215|
51216|     static PSM_StopWatch *All;
51217| };
51218|
51219| #define DECLARE_STOPWATCH(name) static
    | PSM_StopWatch *_PSM_StopWatch_##name = 0;
51220|
51221| #define START_STOPWATCH(name)
    | \
51222|     ((_PSM_StopWatch_##name ? 0 :
    | (_PSM_StopWatch_##name = new PSM_StopWatch(#name))), \
51223|     (_PSM_StopWatch_##name ?
    | _PSM_StopWatch_##name->start() : 0))
51224|
51225| #define PAUSE_STOPWATCH(name)
    | (_PSM_StopWatch_##name->pause())
51226| #define RESET_STOPWATCH(name)
    | (_PSM_StopWatch_##name->reset())
51227| #define DUMP_STOPWATCH(name)
    | (_PSM_StopWatch_##name->dump())
51228| #define STOPWATCH_DUMPALL()
    | (PSM_StopWatch::DumpAll())
51229| #define STOPWATCH_RESETALL()
    | (PSM_StopWatch::ResetAll())
51230| #else /*DEBUG*/
51231| #define DECLARE_STOPWATCH(name)
51232| #define START_STOPWATCH(name) ((void)0)
51233| #define PAUSE_STOPWATCH(name) ((void)0)
51234| #define RESET_STOPWATCH(name) ((void)0)
51235| #define DUMP_STOPWATCH(name) ((void)0)
51236| #define STOPWATCH_DUMPALL() ((void)0)
51237| #define STOPWATCH_RESETALL() ((void)0)
51238| #endif /*DEBUG*/
51239|
51240| void DumpProfileInfo_Thread (void *);
51241|
51242|

```



```

51243|
51244| File Listing: NTFS.h
51245|
51246| typedef CSHORT NODE_TYPE_CODE;
51247| typedef NODE_TYPE_CODE *PNODE_TYPE_CODE;
51248|
51249| #define NTC_UNDEFINED
    | ((NODE_TYPE_CODE)0x0000)
51250|
51251| #define NTFS_NTC_DATA_HEADER
    | ((NODE_TYPE_CODE)0x0700)
51252| #define NTFS_NTC_VCB
    | ((NODE_TYPE_CODE)0x0701)
51253| #define NTFS_NTC_FCB
    | ((NODE_TYPE_CODE)0x0702)
51254| #define NTFS_NTC_DCB
    | ((NODE_TYPE_CODE)0x0703)
51255| #define NTFS_NTC_0703
    | ((NODE_TYPE_CODE)0x0703) // dcb
51256| #define NTFS_NTC_0704
    | ((NODE_TYPE_CODE)0x0704) // root index
51257| #define NTFS_NTC_0705
    | ((NODE_TYPE_CODE)0x0705) // data
51258| #define NTFS_NTC_0706
    | ((NODE_TYPE_CODE)0x0706) // mft
51259| #define NTFS_NTC_CCB
    | ((NODE_TYPE_CODE)0x0707)
51260| #define NTFS_NTC_IRP_CONTEXT
    | ((NODE_TYPE_CODE)0x0708)
51261|
51262| typedef CSHORT NODE_BYTE_SIZE;
51263|
51264| //
51265| // So all records start with
51266| //
51267| // typedef struct _RECORD_NAME {
51268| //     NODE_TYPE_CODE NodeTypeCode;
51269| //     NODE_BYTE_SIZE NodeByteSize;
51270| //     :
51271| // } RECORD_NAME;
51272| // typedef RECORD_NAME *PRECORD_NAME;
51273| //
51274|
51275| #define NodeType(Ptr) (((PNODE_TYPE_CODE)(Ptr)))
51276|
51277| // from ntifs.h
    | *****
    | *
51278|
51279| typedef ULONG LBN;

```

```

51280| typedef LBN *PLBN;
51281|
51282| typedef ULONG VBN;
51283| typedef VBN *PVBN;
51284|
51285|
51286| //
51287| // Every file system that uses the cache manager must
    | have FsContext
51288| // of the file object point to a common fcb header
    | structure.
51289| //
51290| #ifndef FSRTL_FLAG_FILE_MODIFIED
51291| typedef enum _FAST_IO_POSSIBLE {
51292|     FastIoIsNotPossible = 0,
51293|     FastIoIsPossible,
51294|     FastIoIsQuestionable
51295| } FAST_IO_POSSIBLE;
51296|
51297|
51298| #pragma pack()
51299| typedef struct _FSRTL_COMMON_FCB_HEADER {
51300|
51301|     CSHORT NodeTypeCode;
51302|     CSHORT NodeByteSize;
51303|
51304|     //
51305|     // General flags available to FsRtl.
51306|     //
51307|
51308|     UCHAR Flags;
51309|
51310|     //
51311|     // Indicates if fast I/O is possible or if we
    | should be calling
51312|     // the check for fast I/O routine which is found
    | via the driver
51313|     // object.
51314|     //
51315|
51316|     UCHAR IsFastIoPossible; // really type
    | FAST_IO_POSSIBLE
51317|
51318|     //
51319|     // Second Flags Field
51320|     //
51321|
51322|     UCHAR Flags2;
51323|
51324|     //

```

```

51325| // The following reserved field should always be 0
51326| //
51327|
51328| UCHAR Reserved;
51329|
51330| PERESOURCE Resource;
51331|
51332| PERESOURCE PagingIoResource;
51333|
51334| LARGE_INTEGER AllocationSize;
51335| LARGE_INTEGER FileSize;
51336| LARGE_INTEGER ValidDataLength;
51337|
51338| } FSRTL_COMMON_FCB_HEADER;
51339| typedef FSRTL_COMMON_FCB_HEADER
    | *PFSRTL_COMMON_FCB_HEADER;
51340| #endif
51341| //
51342| // Define FsRtl common header flags
51343| //
51344|
51345| #define FSRTL_FLAG_FILE_MODIFIED      (0x01)
51346| #define FSRTL_FLAG_FILE_LENGTH_CHANGED (0x02)
51347| #define FSRTL_FLAG_LIMIT_MODIFIED_PAGES (0x04)
51348|
51349| //
51350| // Following flags determine how the modified page
    | writer should
51351| // acquire the file. These flags can't change while
    | either resource
51352| // is acquired. If neither of these flags is set then
    | the
51353| // modified/mapped page writer will attempt to acquire
    | the paging io
51354| // resource shared.
51355| //
51356|
51357| #define FSRTL_FLAG_ACQUIRE_MAIN_RSRC_EX (0x08)
51358| #define FSRTL_FLAG_ACQUIRE_MAIN_RSRC_SH (0x10)
51359|
51360| //
51361| // This flag will be set by the Cache Manager if a
    | view is mapped
51362| // to a file.
51363| //
51364|
51365| #define FSRTL_FLAG_USER_MAPPED_FILE    (0x20)
51366|
51367| // This flag determines whether there currently is an
    | Eof advance

```

```

51368| // in progress. All such advances must be serialized.
51369| //
51370|
51371| #define FSRTL_FLAG_EOF_ADVANCE_ACTIVE (0x80)
51372|
51373| //
51374| // Flag values for Flags2
51375| //
51376|
51377| //
51378| // If this flag is set, the Cache Manager will allow
    | modified writing
51379| // in spite of the value of FsContext2.
51380| //
51381|
51382| #define FSRTL_FLAG2_DO_MODIFIED_WRITE (0x01)
51383|
51384| //
51385| // The following constants are used to block top level
    | Irp processing when
51386| // (in either the fast io or cc case) file system
    | resources have been
51387| // acquired above the file system, or we are in an Fsp
    | thread.
51388| //
51389|
51390| #define FSRTL_FSP_TOP_LEVEL_IRP      0x01
51391| #define FSRTL_CACHE_TOP_LEVEL_IRP    0x02
51392| #define FSRTL_MOD_WRITE_TOP_LEVEL_IRP 0x03
51393| #define FSRTL_FAST_IO_TOP_LEVEL_IRP  0x04
51394| #define FSRTL_MAX_TOP_LEVEL_IRP_FLAG 0x04
51395|
51396|
51397| #pragma pack()
51398| typedef struct _NTFS_0705 {
51399|
51400|     FSRTL_COMMON_FCB_HEADER VolumeFileHeader; // size
        | = 0x0028
51401|     ULONG Unknown1;                // size
        | = 0x0140
51402|     ULONG Unknown2;
51403|     ULONG Unknown3;
51404|     ULONG Unknown4;
51405|     ULONG Unknown5;
51406|     ULONG Unknown6;
51407|     struct _NTFS_FCB *Fcb;
51408|     struct _NTFS_VCB *Vcb;
51409|     ULONG Unknown8[0x12];
51410|     struct _NTFS_SCB *Scb;
51411|     CHAR Unknown9[0xac];           // 0x0094

```

```

51412|
51413| } NTFS_0705;
51414| typedef NTFS_0705 *PNTFS_0705;
51415|
51416| #pragma pack()
51417| typedef struct _NTFS_0706 {
51418|
51419|     FSRTL_COMMON_FCB_HEADER VolumeFileHeader; // size
        | = 0x0028
51420|     ULONG Unknown1; // size
        | = 0x0168
51421|     ULONG Unknown2;
51422|     ULONG Unknown3;
51423|     ULONG Unknown4;
51424|     ULONG Unknown5;
51425|     ULONG Unknown6;
51426|     struct _NTFS_FCB *Fcb;
51427|     struct _NTFS_VCB *Vcb;
51428|     ULONG Unknown8[0x12];
51429|     struct _NTFS_SCB *Scb;
51430|     CHAR Unknown9[0xd4]; // 0x0094
51431|
51432| } NTFS_0706;
51433| typedef NTFS_0706 *PNTFS_0706;
51434|
51435| #pragma pack()
51436| typedef struct _NTFS_0704 {
51437|
51438|     FSRTL_COMMON_FCB_HEADER VolumeFileHeader; // size
        | = 0x0028
51439|     ULONG Unknown1; // size
        | = 0x0180
51440|     ULONG Unknown2;
51441|     ULONG Unknown3;
51442|     ULONG Unknown4;
51443|     ULONG Unknown5;
51444|     ULONG Unknown6;
51445|     struct _NTFS_FCB *Fcb;
51446|     struct _NTFS_VCB *Vcb;
51447|     ULONG Unknown8[0x12];
51448|     struct _NTFS_SCB *Scb;
51449|     CHAR Unknown9[0xec]; // 0x0094
51450|
51451| } NTFS_0704;
51452| typedef NTFS_0704 *PNTFS_0704;
51453|
51454|
51455| #pragma pack()
51456| typedef struct _NTFS_FCB {
51457|     CSHORT NodeTypeCode; // == 0702

```

```

51458| CSHORT  NodeByteSize;      // == 00b0
51459| PVOID   Vcb;
51460|
51461| ULONG   Unknown1;          // == 00000002
51462| USHORT  Unknown2;          // == 0000
51463| USHORT  Unknown3;          // == 0002
51464|
51465| ULONG   CleanupCount;      // offset 10
51466| ULONG   CloseCount;        // offset 14
51467| ULONG   ReferenceCount;    // offset 18
51468| ULONG   FcbState;          // offset 1c
51469| ULONG   FcbDenyDelete;    // offset 20
51470| ULONG   FcbDeleteFile;     // offset 24
51471|
51472| CHAR    Unknown4[0x1c];
51473|
51474| USHORT  BaseExclusiveCount; // offset 44
51475| USHORT  EaModificationCount; // offset 46
51476|
51477| ULONG   Unknown5;
51478|
51479| ULONG   InfoFlags;         // offset 4c
51480|
51481| CHAR    Unknown6[0x38];
51482|
51483| USHORT  LinkCount;         // offset 88
51484| USHORT  TotalLinks;        // offset 8a
51485| ULONG   CurrentLastAccess; // offset 0
51486|
51487| CHAR    Unknown7[0xa];
51488|
51489| ULONG   CreateSecurityCount; // offset 9c
51490|
51491| ULONG   Unknown8;
51492|
51493| ULONG   DelayedCloseCount; // offset a4
51494|
51495| ULONG   Unknown9;
51496| ULONG   Unknown10;
51497| } NTFS_FCB;
51498| typedef NTFS_FCB *PNTFS_FCB;
51499|
51500| #define FCB_FLAG_NON_PAGED          0x0002
51501| #define FCB_FLAG_DUP_INITIALIZED    0x0008
51502| #define FCB_FLAG_IN_FCB_TABLE       0x0040
51503| #define FCB_FLAG_SYSTEM_FILE        0x0100
51504| #define FCB_FLAG_COMPOUND_INDEX     0x0400
51505|
51506| #pragma pack()
51507| typedef struct _NTFS_VCB {

```

```

51508| CSHORT  NodeTypeCode;      // == 0701
51509| CSHORT  NodeByteSize;      // == 03c8
51510|
51511| ULONG   Unknown1;          //
51512| ULONG   Unknown2;          //
51513| ULONG   Unknown3;          //
51514|
51515| PVOID   MftScb;             // 10
51516| PVOID   Mft2Scb;            // 14
51517| PVOID   LogFileScb;         // 18
51518| PVOID   VolumeDasdScb;      // 1c
51519| PVOID   RootIndexScb;       // 20
51520| PVOID   BitmapScb;          // 24
51521| PVOID   AttributeDefTableScb; // 28
51522| PVOID   UpcaseTableScb;     // 2c
51523| PVOID   BadClusterFileScb;  // 30
51524| PVOID   QuotaTableScb;      // 34
51525| PVOID   OwnerIdTableScb;     // 38
51526| PVOID   MftBitmapScb;       // 3c
51527|
51528| CHAR    Unknown4[0x14];
51529|
51530| ULONG   CleanupCount;        // 54
51531| ULONG   CloseCount;          // 58
51532| ULONG   ReadOnlyCloseCount;  // 5c
51533| ULONG   SystemFileCloseCount; // 60
51534|
51535| CHAR    Unknown5[0x4c];
51536|
51537| PVOID   RootLcb;             // b0
51538|
51539| CHAR    Unknown6[0x314];
51540|
51541| } NTFS_VCB;
51542| typedef NTFS_VCB *PNTFS_VCB;
51543|
51544| // The first 16 files entries are reserved for the
    | following
51545| #define FILE_MFT            0
51546| #define FILE_MFTMirror      1
51547| #define FILE_LogFile        2
51548| #define FILE_Volume         3
51549| #define FILE_AttrDef         4
51550| #define FILE_RootIndex       5
51551| #define FILE_BitMap          6
51552| #define FILE_Boot            7
51553| #define FILE_BadCluster      8
51554| #define FILE_Quota           9
51555| #define FILE_UpCase          10
51556|

```

```

51557| // 11-15 is reserved
51558|
51559| // my types!!!
51560| #define FILE_Reserved    11
51561| #define FILE_OwnerId     12
51562| #define FILE_MFTBitMap    13
51563| #define FILE_RootLcb      14
51564| #define FILE_Other        15
51565|
51566| #pragma pack()
51567|
51568|
51569|
51570| File Listing: ONDISK.h
51571|
51572| #pragma pack(1)
51573| typedef struct sPartRec {
51574|     UCHAR Bootable;
51575|     UCHAR BeginHead;
51576|     UCHAR BeginSector;
51577|     UCHAR BeginCyl;
51578|     UCHAR SystemType;
51579|     UCHAR EndHead;
51580|     UCHAR EndSector;
51581|     UCHAR EndCyl;
51582|     ULONG StartSector;
51583|     ULONG NumberOfSectors;
51584| } tPartRec;
51585|
51586| #pragma pack(1)
51587| typedef struct sMBR {
51588|     UCHAR    Code[0x1b9];
51589|     ULONG    SerialNumber;
51590|     UCHAR    Unknown1;
51591|     tPartRec Partition[4];
51592|     USHORT   Signature;
51593| } tMBR;
51594|
51595| #pragma pack(1)
51596| typedef struct _NTFS_BOOT_SECTOR {
51597|     UCHAR    Jmp[3];                //
51598|     | 0
51599|     CHAR     OemId[8];              //
51600|     | 3
51601|     USHORT   BytesPerSector;        //
51602|     | 11
51603|     UCHAR    SectorsPerCluster;     //
51604|     | 13
51605|     USHORT   ReservedSectors;      //
51606|     | 14 - not used in NT (0)

```



```

51602|  UCHAR   NumberOfFats;           //
      | 16 - not used in NT (0)
51603|  USHORT   RootDirectory;          //
      | 17 - not used in NT (0)
51604|  USHORT   SmallSectors;          //
      | 19 - not used in NT (0)
51605|  UCHAR   MediaID;                //
      | 21
51606|  USHORT   SectorsPerFat;          //
      | 22 - not used in NT (0)
51607|  USHORT   SectorsPerTrack;        //
      | 24
51608|  USHORT   Heads;                  //
      | 26
51609|  ULONG    HiddenSectors;          //
      | 28
51610|  ULONG    LargeSectors;            //
      | 32
51611|  UCHAR   Drive;                   //
      | 36
51612|  UCHAR   DirtyVolume;             //
      | 37
51613|
51614|  USHORT   Reserved;                //
      | 38
51615|  ULARGE_INTEGER TotalSectors;        //
      | 40
51616|  ULARGE_INTEGER ClusterToMFT;        //
      | 48
51617|  ULARGE_INTEGER ClustersToMFTMirror; //
      | 56
51618|  ULONG    ClustersPerFRS;           //
      | 64
51619|  ULONG    ClustersPerIndexBlock;     //
      | 68
51620|  ULARGE_INTEGER SerialNumber;        //
      | 72
51621|  ULONG    SectorChecksum;           //
      | 80
51622|  ULONG    Unknown1;                 //
      | 84
51623|  ULONG    Unknown2;                 //
      | 88
51624|  UCHAR   Unknown3;                 //
      | 92
51625|  UCHAR   Code[0x1a1];              //
      | 93
51626|  USHORT   Signature;                //
      | 510 - aa55
51627| } NTFS_BOOT_SECTOR;

```

```

51628| typedef NTFS_BOOT_SECTOR *PNTFS_BOOT_SECTOR;
51629|
51630| #pragma pack(1)
51631| typedef struct _FAT_BOOT_SECTOR {
51632|  UCHAR   Jmp[3];                //
    | 0
51633|  CHAR    OemId[8];              //
    | 3
51634|  USHORT  BytesPerSector;        //
    | 11
51635|  UCHAR   SectorsPerCluster;     //
    | 13
51636|  USHORT  ReservedSectors;       //
    | 14
51637|  UCHAR   NumberOfFats;          //
    | 16
51638|  USHORT  RootDirectory;         //
    | 17
51639|  USHORT  SmallSectors;          //
    | 19
51640|  UCHAR   MediaID;              //
    | 21
51641|  USHORT  SectorsPerFat;         //
    | 22
51642|  USHORT  SectorsPerTrack;       //
    | 24
51643|  USHORT  Heads;                //
    | 26
51644|  ULONG   HiddenSectors;         //
    | 28
51645|  ULONG   LargeSectors;          //
    | 32
51646|  UCHAR   Drive;                //
    | 36
51647|  UCHAR   DirtyVolume;          //
    | 37
51648|
51649|  UCHAR   BootSignature;         //
    | 38 - 29 means the next are valid
51650|  ULONG   VolumeID;             //
    | 39
51651|  CHAR    VolumeLabel[0xb];      //
    | 43
51652|  CHAR    FileSysType[8];        //
    | 54
51653|  UCHAR   Code[0x1c0];          //
    | 62
51654|  USHORT  Signature;            //
    | 510 - aa55
51655| } FAT_BOOT_SECTOR;

```

```

51656| typedef FAT_BOOT_SECTOR *PFAT_BOOT_SECTOR;
51657|
51658| #pragma pack(1)
51659| typedef struct _FAT32_BOOT_SECTOR {
51660|     UCHAR    Jmp[3];                //
        | 0
51661|     CHAR     OemId[8];              //
        | 3
51662|
51663|     USHORT   BytesPerSector;        //
        | 11
51664|     UCHAR     SectorsPerCluster;    //
        | 13
51665|     USHORT   ReservedSectors;      //
        | 14
51666|     UCHAR     NumberOfFats;         //
        | 16
51667|     USHORT   RootEntries;           //
        | 17
51668|     USHORT   SmallSectors;          //
        | 19
51669|     UCHAR     MediaID;              //
        | 21
51670|     USHORT   SectorsPerFat;         //
        | 22
51671|     USHORT   SectorsPerTrack;      //
        | 24
51672|     USHORT   Heads;                //
        | 26
51673|     ULONG    HiddenSectors;        //
        | 28
51674|     ULONG    LargeSectors;         //
        | 32
51675|     ULONG    LargeSectorsPerFat;    //
        | 36
51676|     USHORT   ExtendedFlags;        //
        | 40
51677|     USHORT   FsVersion;            //
        | 42
51678|     ULONG    RootDirFirstCluster;  //
        | 44
51679|     USHORT   FsInfoSector;         //
        | 48
51680|     USHORT   BackupBootSector;     //
        | 50
51681|     UCHAR     Reserved[12];        //
        | 52
51682|
51683|     UCHAR     Drive;               //
        | 64

```

```

51684|  UCHAR   DirtyVolume;           //
      | 65
51685|
51686|  UCHAR   BootSignature;          //
      | 66 - 29 means the next are valid
51687|  ULONG   VolumeID;               //
      | 67
51688|  CHAR    VolumeLabel[0xb];        //
      | 71
51689|  CHAR    FileSysType[8];          //
      | 82
51690|  UCHAR   Code[0x1a4];            //
      | 90
51691|  USHORT  Signature;              //
      | 510 - aa55
51692| } FAT32_BOOT_SECTOR;
51693| typedef FAT32_BOOT_SECTOR *PFAT32_BOOT_SECTOR;
51694|
51695| //
51696| // Define the FAT32 FsInfo sector.
51697| //
51698|
51699| #pragma pack(1)
51700| typedef struct _FSINFO_SECTOR {
51701|     ULONG SectorBeginSignature;      //
      | offset = 0x000 0
51702|     UCHAR ExtraBootCode[480];        //
      | offset = 0x004 4
51703|     ULONG FsInfoSignature;          //
      | offset = 0x1e4 484
51704|     ULONG FreeClusterCount;          //
      | offset = 0x1e8 488
51705|     ULONG NextFreeCluster;          //
      | offset = 0x1ec 492
51706|     UCHAR Reserved[12];             //
      | offset = 0x1f0 496
51707|     ULONG SectorEndSignature;        //
      | offset = 0x1fc 508
51708| } FSINFO_SECTOR, *PFSINFO_SECTOR;
51709|
51710| #define FSINFO_SECTOR_BEGIN_SIGNATURE 0x41615252
      | // AaRR
51711| #define FSINFO_SECTOR_END_SIGNATURE 0xAA550000
51712|
51713| #define FSINFO_SIGNATURE 0x61417272
      | // aArr
51714|
51715|
51716| #pragma pack(1)
51717| typedef struct _FAT_DIR_ENTRY {

```

```

51718|  CHAR  Name[8];
51719|  CHAR  Ext[3];
51720|  UCHAR  Attributes;
51721|  UCHAR  Reserved[0xa];
51722|  USHORT  Time;
51723|  USHORT  Date;
51724|  USHORT  StartCluster;
51725|  ULONG  FileSize;
51726| } FAT_DIR_ENTRY;
51727| typedef FAT_DIR_ENTRY *PFAT_DIR_ENTRY;
51728|
51729|
51730| #pragma pack()
51731|
51732|
51733|
51734| File Listing: PASSTHRU.cpp
51735|
51736| #include "precomp.h"
51737|
51738|
51739| /*-----
    | -----*/
51740| NTSTATUS
51741| PManPassThru(
51742|     IN PDEVICE_OBJECT DeviceObject,
51743|     IN PIRP Irp
51744| )
51745|
51746| /*++
51747|
51748| Routine Description:
51749|
51750| This is the driver entry point for read requests
51751| to disks to which the PMan driver has attached.
51752| This driver collects statistics and then sets a
    | completion
51753| routine so that it can collect additional
    | information when
51754| the request completes. Then it calls the next
    | driver below
51755| it.
51756|
51757| Arguments:
51758|
51759| DeviceObject
51760| Irp
51761|
51762| Return Value:
51763|

```

```

51764| NTSTATUS
51765|
51766| --*/
51767|
51768| {
51769|
51770| #if 0
51771|     TRACE( TRACE_PASSTHRU,
51772|         | currentIrpStack->Parameters.Read.ByteOffset.HighPart,
51773|         | currentIrpStack->Parameters.Read.ByteOffset.LowPart,
51774|         currentIrpStack->Parameters.Read.Length,
51775|         currentIrpStack->Parameters.Read.Key,
51776|         "");
51777| #endif
51778|
51779| #ifdef DEBUG
51780|     if(DebugPrints) {
51781|         char Buff[30]={0};
51782|         PIO_STACK_LOCATION currentIrpStack =
51783|         | IoGetCurrentIrpStackLocation(Irp);
51784|         sprintf(Buff,"PassThru %d,%d: ",
51785|             currentIrpStack->MajorFunction,
51786|             currentIrpStack->MinorFunction
51787|         );
51788|         File_PrintOneLiner(Buff,DeviceObject,Irp);
51789|     }
51790| #endif
51791|
51792|     if(PsmGetObjectType(DeviceObject)==OBJECT_INTERNAL)
51793|     | {
51794|         Debug(DEBUG_PASSTHRU | DEBUG_ERROR,("Error!
51795|         | Device is PSMAN Object, Failing request\n"));
51796|
51797|         Irp->IoStatus.Status =
51798|         | STATUS_INVALID_DEVICE_REQUEST;
51799|         Irp->IoStatus.Information = 0;
51800|         IoCompleteRequest(Irp, IO_NO_INCREMENT);
51801|         InterlockedDecrement(
51802|         | (PLONG)&OutstandingRequests );
51803|         return STATUS_INVALID_DEVICE_REQUEST;
51804|     } else
51805|
51806|     | if(PsmGetObjectType(DeviceObject)==OBJECT_VIRTUALDISK)
51807|     | {
51808|         Debug(DEBUG_PASSTHRU | DEBUG_ERROR,("Error!
51809|         | Device is vdisk Object, Failing request\n"));
51810|

```

```

51804|     Irp->IoStatus.Status =
        | STATUS_INVALID_DEVICE_REQUEST;
51805|     Irp->IoStatus.Information = 0;
51806|     IoCompleteRequest(Irp, IO_NO_INCREMENT);
51807|     InterlockedDecrement(
        | (PLONG)&OutstandingRequests );
51808|     return STATUS_INVALID_DEVICE_REQUEST;
51809| } else
51810|
        | if(PsmGetObjectTypes(DeviceObject)==OBJECT_FS_OBJECT) {
51811|     Debug(DEBUG_PASSTHRU | DEBUG_ERROR,("Error!
        | Device is File System Object , Failing request\n"));
51812|
51813|     Irp->IoStatus.Status =
        | STATUS_INVALID_DEVICE_REQUEST;
51814|     Irp->IoStatus.Information = 0;
51815|     IoCompleteRequest(Irp, IO_NO_INCREMENT);
51816|     InterlockedDecrement(
        | (PLONG)&OutstandingRequests );
51817|     return STATUS_INVALID_DEVICE_REQUEST;
51818| }
51819|
51820| #ifdef DEBUG
51821|     if(DebugPrints) {
51822|         // Copy current stack to next stack.
51823|         IoCopyCurrentIrpStackLocationToNext(Irp);
51824|
51825|         // Set completion routine callback.
51826|         /*lint -save -e506 -e774 */
51827|         IoSetCompletionRoutine(Irp,
51828|                                PSMAnIoCompletionDevice,
51829|                                DeviceObject,
51830|                                TRUE,
51831|                                TRUE,
51832|                                TRUE);
51833|         /*lint -restore */
51834|     } else {
51835|         IoSkipCurrentIrpStackLocation( Irp );
51836|     }
51837| #else
51838|     IoSkipCurrentIrpStackLocation( Irp );
51839| #endif
51840|
51841|     // Return the results of the call to the disk
        | driver.
51842|
51843|     PFILTERED_EXTENSION FilteredExt =
        | GetFilteredExtension(DeviceObject);
51844|     return IoCallDriver
        | (FilteredExt->TargetDeviceObject, Irp);

```

```

51845|
51846| } // end PSMANPassThru()
51847|
51848|
51849| /*-----*
    | -----*/
51850| NTSTATUS
51851| PSMANIoCompletionDevice(
51852|     IN PDEVICE_OBJECT DeviceObject,
51853|     IN PIRP           Irp,
51854|     IN PVOID          Context
51855| )
51856|
51857| /*++
51858|
51859| Routine Description:
51860|
51861|     This routine will get control from the system at
    | the completion of an IRP.
51862|     It will calculate the difference between the time
    | the IRP was started
51863|     and the current time, and decrement the queue
    | depth.
51864|
51865|     IRQL <= DISPATCH_LEVEL Assume running at
    | DISPATCH_LEVEL!!!
51866|     as it will vary depending on what the higher
    | level driver is doing
51867|
51868| Arguments:
51869|
51870|     DeviceObject - for the IRP.
51871|     Irp          - The I/O request that just completed.
51872|     Context      - Not used.
51873|
51874| Return Value:
51875|
51876|     The IRP status.
51877|
51878| --*/
51879|
51880| {
51881|
51882|     NOT_REFERENCED(Context);
51883| #if 0
51884|     TRACE( TRACE_PASSTHRU_COMP,
51885|
    | irpStack->Parameters.Read.ByteOffset.HighPart,
51886|
    | irpStack->Parameters.Read.ByteOffset.LowPart,

```



```

51887|         irpStack->Parameters.Read.Length,
51888|         irpStack->Parameters.Read.Key,
51889|         "");
51890| #endif
51891|
51892| #ifdef DEBUG
51893|     if(DebugPrints) {
51894|         File_PrintOneLiner("PassThru Done:
51895|         | ",DeviceObject,Irp);
51896|     }
51897| #endif
51898|     if (Irp->PendingReturned) {
51899|         IoMarkIrpPending(Irp);
51900|     }
51901|
51902|     return STATUS_SUCCESS;
51903|
51904| } // PSManIoCompletionDevice
51905|
51906|
51907|
51908| File Listing: PASSTHRU.h
51909|
51910| NTSTATUS
51911| PSManPassThru(
51912|     IN PDEVICE_OBJECT DeviceObject,
51913|     IN PIRP Irp
51914| );
51915|
51916| NTSTATUS
51917| PSManIoCompletionDevice(
51918|     IN PDEVICE_OBJECT DeviceObject,
51919|     IN PIRP Irp,
51920|     IN PVOID Context
51921| );
51922|
51923|
51924|
51925| File Listing: perapi.cpp
51926|
51927| #include "precomp.h"
51928|
51929| //-----
51930| | -----
51931| void __cdecl OurFree ( void *p )
51932| {
51933|     if ( p ) {
51934|         MemFreePool(p);

```

```

51935|    }
51936| }
51937|
51938| //-----
    | -----
51939|
51940| PVOID __cdecl OurMalloc (
51941|     ULONG x,
51942|     ULONG id,
51943|     POOL_TYPE pool )
51944| {
51945|     PVOID buffer = MemAllocatePoolWithTag (pool, x,
        | id);
51946|     if ( buffer ) {
51947|         RtlZeroMemory (buffer, x);
51948|     }
51949|
51950|     return buffer;
51951| }
51952|
51953| //-----
    | -----
51954|
51955| void DeletableObject::operator delete (void *p)
51956| {
51957|     OurFree(p);
51958| }
51959|
51960| //-----
    | -----
51961|
51962| void * __cdecl operator new ( size_t size )
51963| {
51964|     return OurMalloc ( size, 'ppc0', NonPagedPool );
51965| }
51966|
51967| //-----
    | -----
51968|
51969| ErrorCode
51970| Dictionary::Open (
51971|     ULONG         flags,
51972|     Dictionary *   &pDictionary )
51973| {
51974|     ErrorCode error = STATUS_SUCCESS;
51975|     pDictionary = 0;
51976|
51977|     if ( flags & DICT_FLAG_NONPERSISTENT ) {
51978|         pDictionary = new TemporaryDictionary();
51979|     } else if ( flags & DICT_FLAG_PERSISTENT ) {

```

```

51980|     pDictionary = new PersistentDictionary();
51981| } else {
51982|     error = STATUS_INVALID_PARAMETER;
51983| }
51984|
51985| if ( error==STATUS_SUCCESS && !pDictionary ) {
51986|     error = STATUS_NO_MEMORY;
51987| }
51988|
51989| return error;
51990| }
51991|
51992| //-----
| -----
51993|
51994| ErrorCode
51995| Dictionary::Destroy (
51996|     Dictionary * &pDictionary )
51997| {
51998|     ErrorCode error = pDictionary->close();
51999|     delete pDictionary;
52000|     pDictionary = 0;
52001|     return error;
52002| }
52003|
52004| /*--- end of file perapi.cpp ---*/
52005|
52006|
52007|
52008| File Listing: perapi.h
52009|
52010| typedef const void *PCVOID;
52011| typedef NTSTATUS ErrorCode;
52012|
52013| #define DICT_FLAG_NONPERSISTENT    0x1
52014| #define DICT_FLAG_PERSISTENT      0x2
52015| #define DICT_FLAG_PERSERVE_MEMORY 0x4
52016| #define DICT_FLAG_PERSERVE_DISK   0x8
52017| #define DICT_FLAGS_USE_VLM        0x10
52018| #define DICT_FLAGS_USE_PAE        0x20
52019| #define DICT_FLAGS_REOPEN_EXISTING 0x40
52020|
52021| // for insert functions
52022| #define DICT_FLAG_REPLACE_DUPS    0x1
52023|
52024| // for insert and search
52025| #define DICT_FLAG_VIRTUAL_IO      0x8000
52026|
52027| #define DICT_ACCESS_READONLY      (GENERIC_READ)
52028| #define DICT_ACCESS_READWRITE    (GENERIC_READ |

```

```

    | GENERIC_WRITE)
52029| #define DICT_ACCESS_QUERY      (READ_CONTROL)
52030| #define DICT_ACCESS_CHANGE      (WRITE_DAC |
    | WRITE_OWNER | READ_CONTROL)
52031| #define DICT_ACCESS_DELETE      (DELETE)
52032| #define DICT_ACCESS_WRITE_DAC    (WRITE_DAC)
52033| #define DICT_ACCESS_OWNER        (WRITE_OWNER)
52034| #define DICT_ACCESS_SYNCHRONIZE  (SYNCHRONIZE)
52035| #define DICT_ACCESS_STANDARD_RIGHTS (GENERIC_ALL)
52036|
52037| #define GranulePart HighPart
52038| #define SnapShotPart LowPart
52039|
52040|
52041| typedef struct _FILTERED_EXTENSION
    | *PFILTERED_EXTENSION;
52042| typedef struct sDictSiblingInfo {
52043|     char Opaque[256];
52044| } tDictSiblingInfo, *pDictSiblingInfo;
52045|
52046| void * __cdecl operator new ( size_t size );
52047|
52048|
52049| // We created the class DeletableObject because we
    | weren't having
52050| // much luck getting the compiler to pay any attention
    | to our
52051| // global function ::operator delete. (Compiler
    | refuses to generate code.)
52052|
52053| class DeletableObject
52054| {
52055| public:
52056|     void __cdecl operator delete ( void *p );
52057| };
52058|
52059|
52060| enum DictionaryInformation {
52061|     DictBasicInformation,
52062|     DictStandardInformation,
52063|     DictSecurityInformation,
52064|     DictAllInformation
52065| };
52066|
52067|
52068|
52069| class Dictionary: public DeletableObject
52070| {
52071| public:
52072|     virtual ~Dictionary() { cleanup(); }

```

```

52073|
52074| virtual ErrorCode initialize ( ULONG flags ) = 0;
52075| virtual ErrorCode cleanup() { return
    | STATUS_SUCCESS; }
52076| virtual ErrorCode GetOutParams( WCHAR *Cache ) = 0;
52077|
52078|
52079| virtual ErrorCode searchAndDeleteSingle(
52080|     PFILTERED_EXTENSION devExt,
52081|     ULARGE_INTEGER      sector ) = 0;
52082|
52083|
    | /*-----
    | -----
52084|     searchMultiple
52085|     If one or more of the sectors exist in the
    | dictionary
52086|     Returns:
52087|         Dictionary::STATUS_SUCCESS
52088|         Dictionary::INSUFFICIENT_RESOURCES
52089|         Dictionary::INSUFFICIENT_DISK_SPACE
52090|         Dictionary::OUT_OF_MEMORY
52091|         etc...
52092|     Starting - Starting Sector
52093|     Count    - Number of sectors to do (0 is
    | allowed)
52094|     CountDid - Number of sectors found (0 is
    | allowed)
52095|     Bitmap   - if NULL no data returned
52096|                else if a bit is set, the sector
    | corresponding to that
52097|                bit has been stored in the
    | dictionary
52098|
52099| */
52100| virtual ErrorCode searchMultiple(
52101|     PFILTERED_EXTENSION devExt,
52102|     ULARGE_INTEGER      starting,
52103|     ULONG                count,
52104|     ULONG                &countDid,
52105|     PRTL_BITMAP          bitMap,
52106|     ULARGE_INTEGER      dataSize,
52107|     PVOID                virtualDataPointer,
52108|     ULONG                flags ) = 0;
52109|
52110|
    | /*-----
    | -----
52111|     searchAndInsertMultiple
52112|     If one or more of the sectors exist in the

```

```

    | dictionary, if they do
52113|     not, one or more are added
52114|
52115|     Returns:
52116|         Dictionary::STATUS_SUCCESS
52117|         Dictionary::INSUFFICIENT_RESOURCES
52118|         Dictionary::INSUFFICIENT_DISK_SPACE
52119|         Dictionary::OUT_OF_MEMORY
52120|         etc...
52121|     Starting - Starting Sector
52122|     Count - Number of sectors to do (0 is
    | allowed)
52123|     CountDid - Number of sectors __added__ to the
    | dictionary (0 is allowed)
52124|     Bitmap - if NULL no data returned
52125|             else if a bit is set, the sector
    | corresponding to that
52126|             bit has been stored in the
    | dictionary else it was
52127|             already in the dictionary
52128|
52129|     DataSize - Number of bytes in the data pointer
52130|     VirtualDataPointer - Pointer to data to be
    | added to the dictionary
52131| */
52132| virtual ErrorCode searchAndInsertMultiple(
52133|     PFILTERED_EXTENSION devExt,
52134|     ULARGE_INTEGER starting,
52135|     ULONG count,
52136|     ULONG &countDid,
52137|     PRTL_BITMAP bitMap,
52138|     ULARGE_INTEGER dataSize,
52139|     PCVOID dataPointer,
52140|     pDictSiblingInfo siblingInfo,
52141|     ULONG Flags ) = 0;
52142|
52143|
    | /*-----
    | -----
52144|     Open
52145|     Opens the dictionary and allocates any needed
    | resources (memory,
52146|     disk, etc...)
52147|
52148|     Returns:
52149|         Dictionary::STATUS_SUCCESS
52150|         Dictionary::INSUFFICIENT_RESOURCES
52151|         Dictionary::INSUFFICIENT_DISK_SPACE
52152|         Dictionary::OUT_OF_MEMORY
52153|         etc...

```

```

52154|      Path - Path and options for dictionary some
      | examples could be
52155|      file://servername/diskname/cachefilename
52156|      ftp://servername/directory/cachefilename
52157|      memory://servername
52158|      etc...
52159|
52160|      Options are seperated via semicolons, multi
      | options for 1 option
52161|      is seperated via comma's some options could
      | be
52162|
52163|      MaxDiskSpace=80MB
52164|      MaxMemory=90MB
52165|      UseDisks=C,D
52166|      etc...
52167|
52168|      so a full path would look like
52169|
52170|      | "file://bart/c:/cache/psm.$$$?MaxDiskSpace=80MB;UseDisks
      | =C,D;MaxMemory=80MB"
52171|
52172|
52173|
52174|      Flags - Dictionary flags (DICT_FLAG_PERSISTENT,
      | etc...)
52175|      Access - Access rights to dictionary
52176|      pDictionary - Returned handle
52177|      */
52178|      static ErrorCode Open (
52179|          ULONG      flags,
52180|          Dictionary * &pDictionary );
52181|
52182|
      | /*-----
      | -----
52183|      Destroy
52184|      First closes the dictionary object, then
      | deallocates memory for it.
52185|
52186|      Returns:
52187|      Dictionary::STATUS_SUCCESS
52188|      Dictionary::INVALID_PARAMETER
52189|
52190|      pDictionary - Passed in as pointer to
      | dictionary.
52191|      Will be set to NULL upon return.
52192|      */
52193|      static ErrorCode Destroy (

```

```

52194|     Dictionary *  &pDictionary );
52195|
52196|
52197|
52198|     | /*-----
52199|     | -----
52200|     close
52201|     Closes dictionary.
52202|     This dictionary can be reopened without any
52203|     | data loss. Data is
52204|     flushed to disk before being closed. If
52205|     | nonpersistent (should we
52206|     call this 'temporary'?) the data files will be
52207|     | orphaned and deleted
52208|     upon bootup. Example scenario: The computer
52209|     | crashes while a
52210|     nonpersistent snapshot is active. When the
52211|     | machine reboots, our
52212|     software notices the orphaned cache files and
52213|     | erases them to free up
52214|     the wasted disk space.
52215|
52216|     This will force data to be flushed to disk, and
52217|     | close the dictionary
52218|     in a consistent state.
52219|
52220|     */
52221|     virtual ErrorCode close() = 0;
52222|
52223|
52224|     | /*-----
52225|     | -----
52226|     destroy
52227|
52228|     Closes and deletes the dictionary for disk.
52229|     | This dictionary can not
52230|     be reopened.
52231|
52232|     */
52233|     virtual ErrorCode destroy() = 0;
52234|
52235|
52236|
52237|     | /*-----
52238|     | -----
52239|     reset
52240|
52241|     Causes the dictionary to forget all the sectors
52242|     | it has cached. This
52243|     is a very quick operation, perhaps involving

```



```

    | only a few "pointer"
52229|     operations which can be rolled back as just
    | another transaction (if
52230|     we can save the tiny amount of data that gets
    | overwritten during the
52231|     reset.)
52232|
52233|     */
52234|     virtual ErrorCode reset() = 0;
52235|
52236|
52237|     virtual ULONG GetSequenceNumber() const { return
    | 0xffffffff; }
52238|     virtual __int64 GetSnapShotTime() const { return
    | __int64(0); }
52239|
52240| protected:
52241|     // !!! Put any code/data here that should be
    | inherited by ALL derived classes
52242|     // (e.g. TemporaryDictionary and
    | PersistentDictionary), but that you want
52243|     // to be invisible to code outside this class
    | hierarchy.
52244| };
52245|
52246| typedef Dictionary *pDictionary;
52247|
52248|
52249| //-----
    | ----
52250| // tRevertInfo defines how to initiate/complete a
    | revert
52251| // operation at boot time...
52252| //-----
    | ----
52253| typedef struct sRevertInfo {
52254|     ULONG        SnapShotSequenceNumber;    //
    | identify snapshot to revert, or 0 if none
52255|     ULONG        Reserved2;                // used
    | to be 'VolumeldInProgress'
52256|     ULARGE_INTEGER LastKnownGranuleFinished; //
    | granule which we know was finished
52257|     ULONG        RevertFlags;
52258|     ULONG        Reserved3;
52259|     LARGE_INTEGER SnapShotTime;
52260|     ULONG        Reserved [2];
52261| } tRevertInfo, *pRevertInfo;
52262|
52263|
52264| //-----

```

```

| -----
52265| #define GRANULE_SIZE          (65536)
52266| //#define GRANULE_SIZE        (1024)
52267| #define SECTORS_PER_GRANULE     (GRANULE_SIZE /
| SectorSize)
52268| #define ROUND_UNITS_UP(c,s) (((c)+((s)-1)) / (s))
52269| #define ROUND_UP(c,s) (((((c)+((s)-1)) / (s)))*(s))
52270| #define ROUND_DOWN(c,s) (((c) / (s))*(s))
52271|
52272| const ULONG  DEFAULT_HEADER_BLOCKS_TO_SAVE = 2;
52273| const ULONG  MAX_SNAPSHOTS_PER_HEADER_FILE = 1024;
52274| const ULONG  HEADER_SIZE_IN_RELEASE_2     = 1179648;
52275|
52276| #define PSM_DO_NOT_DO_FREE_SPACE  (0x01)
52277|
52278| class tVirginMap;
52279|
52280| void DeletePsmFilesOnVolume ( PFILTERED_EXTENSION );
| // for use by 'resetpsm'
52281| NTSTATUS Rebuild_DismountAllVolumes( PDEVICE_OBJECT
| Volume, BOOLEAN DisableMounts );
52282| NTSTATUS Rebuild_ReenableVolumeMounts ( PDEVICE_OBJECT
| Volume );
52283|
52284| #ifdef DEBUG
52285|     #define DEBUG_VALIDATE_DIFF_GRANULES  0
52286| #else
52287|     #define DEBUG_VALIDATE_DIFF_GRANULES  0
52288| #endif /*DEBUG*/
52289|
52290| class PersistentDictionary: public Dictionary
52291| {
52292| private:
52293|     #include "header.h"
52294|
52295| public:
52296|     PersistentDictionary();
52297|     ~PersistentDictionary();
52298|
52299|     virtual ErrorCode initialize ( ULONG Flags );
52300|     virtual ErrorCode cleanup(void);
52301|
52302|     static ErrorCode InitClass( ULONG Stage,
| tOpenTransactionInternal *In, PVOID AbortEvent );
52303|
52304|     static ErrorCode DeinitClass( );
52305|     static ErrorCode GetVolumeSpaceUsed( PDEVICE_OBJECT
| Volume, ULONG &At, ULONG &High, ULONG &Used );
52306|
52307|     virtual ErrorCode GetOutParams( WCHAR *Cache );

```

```

52308|
52309| virtual ErrorCode searchAndDeleteSingle(
52310|     PFILTERED_EXTENSION devExt,
52311|     ULARGE_INTEGER      sector );
52312|
52313| virtual ErrorCode searchMultiple(
52314|     PFILTERED_EXTENSION devExt,
52315|     ULARGE_INTEGER      starting,
52316|     ULONG               count,
52317|     ULONG               &countDid,
52318|     PRTL_BITMAP         bitMap,
52319|     ULARGE_INTEGER      dataSize,
52320|     PVOID               virtualDataPointer,
52321|     ULONG               flags );
52322|
52323| virtual ErrorCode searchAndInsertMultiple(
52324|     PFILTERED_EXTENSION devExt,
52325|     ULARGE_INTEGER      starting,
52326|     ULONG               count,
52327|     ULONG               &countDid,
52328|     PRTL_BITMAP         bitMap,
52329|     ULARGE_INTEGER      dataSize,
52330|     PCVOID              dataPointer,
52331|     pDictSiblingInfo    siblingInfo,
52332|     ULONG               Flags);
52333|
52334| virtual ErrorCode close(void);
52335| virtual ErrorCode destroy(void);
52336| virtual ErrorCode reset(void);
52337|
52338| static ErrorCode GetDictionaryForVolume( PVOID
    | Volume, pDictionary &Dictionary );
52339| static ErrorCode GetDictionaryForVolume( PVOID
    | Volume, pDictionary &Dictionary, ULONG Index );
52340|
52341| ErrorCode SetVolume( PDEVICE_OBJECT AVolume,
    | skSnapshotMaster *MasterSnapshot, ULONG
    | SnapshotSequence );
52342| ErrorCode SetSnapshotInfo( skSnapshotMaster *Master
    | );
52343| static ErrorCode SetInfo(
    | tOpenTransactionInternal *In );
52344| ULONG IsCacheWarningThresholdReached();
52345| ULONG IsCacheSnapshotCreationThresholdReached();
52346| // static ULONG      CacheFullThresholdPercent;
52347| // static ULONG      CacheWarningInterval;
52348| // static ULONG      CacheWarningThresholdPercent;
52349| // FIXFIXFIX This is a temporary flag to prevent
    | caching during rebuild - to be replaced
52350| // with a holding pen to save the writes till after

```

| rebuild is through.

```
52351| NTSTATUS PersistentDictionary::ProcessCachingMap(  
    | PRTL_BITMAP *CachingBitMap, const WCHAR  
    | *VirtualVolName, const WCHAR *LiveVolName );  
52352| NTSTATUS SetFreeSpaceStatus( ULARGE_INTEGER Sector,  
    | ULONG Count, BOOLEAN Set );  
52353| BOOLEAN NeedsCaching( ULARGE_INTEGER Sector, ULONG  
    | Count );  
52354| BOOLEAN NeedMap(void);  
52355| static ErrorCode  
    | UpdateCacheFileSizes(PDEVICE_OBJECT Volume);  
52356| static ErrorCode BeginUpdate(void);  
52357| static ErrorCode EndUpdate(void);  
52358| static ULONG QueryMaxNumUserSnapShots();  
52359| NTSTATUS RemoveVirtualWrites(void);  
52360| virtual ULONG GetSequenceNumber() const;  
52361| virtual __int64 GetSnapShotTime() const;  
52362| static BOOLEAN DoFreeSpaceChecks(void);  
52363|  
52364| virtual PRTL_BITMAP GetVolumeCachingMap( ULONG Map  
    | = 0 );  
52365| virtual void  
    | PersistentDictionary::SetVolumeCachingMap( ULONG Map ,  
    | PRTL_BITMAP BitMap );  
52366| virtual ULONG GetVolumeClusterSize(void);  
52367|  
52368| ULONG GetSnapShotFlags(void);  
52369| ULONG IsReadOnly() { return PSM_SS_IsReadOnly  
    | ((unsigned char) GetSnapShotFlags()); }  
52370| ULONG IsReadWrite() { return PSM_SS_IsReadWrite  
    | ((unsigned char) GetSnapShotFlags()); }  
52371| ULONG IsPersistent() { return PSM_SS_IsPersistent  
    | ((unsigned char) GetSnapShotFlags()); }  
52372| ULONG IsTemporary() { return PSM_SS_IsTemporary  
    | ((unsigned char) GetSnapShotFlags()); }  
52373|  
52374| static void GetRevertBootInfo ( tRevertInfo & );  
52375| static NTSTATUS SetRevertBootInfo ( PDEVICE_OBJECT  
    | Volume, const tRevertInfo & );  
52376| static NTSTATUS SaveHeader(PDEVICE_OBJECT Volume);  
52377| static void GetCacheThresholds( PDEVICE_OBJECT  
    | Volume, ULONG &Warn, ULONG &Full, ULONG &Interval,  
    | ULONG &FullPercent, ULONG &FullAction );  
52378| static NTSTATUS LoadSnapShotsForVolume(  
    | PDEVICE_OBJECT Volume, BOOLEAN Rebuild=TRUE, PVOID  
    | AbortEvent=NULL );  
52379| static NTSTATUS UnloadSnapShotsForVolume(  
    | PDEVICE_OBJECT Volume, ULONG OpenCloseOwned=FALSE );  
52380| static NTSTATUS MiniUnloadSnapShotsForVolume(  
    | PDEVICE_OBJECT Volume, ULONG OpenCloseOwned=FALSE );
```

```

52381| static NTSTATUS CloseFilesForVolume( PDEVICE_OBJECT
| Volume );
52382| static NTSTATUS RebuildSnapShotsForVolume(
| PDEVICE_OBJECT Volume, BOOLEAN Rebuild=TRUE, PVOID
| AbortEvent=NULL );
52383| static NTSTATUS RetrieveDirectIOMaps(
| PDEVICE_OBJECT Volume, BOOLEAN
| CheckClusterDataBase=TRUE );
52384| static NTSTATUS StoreClustersOfFiles(
| PDEVICE_OBJECT Volume );
52385| static void Part2OfRebuildForVolume( PVOID Volume
| );
52386| static void SetSystemReady(void);
52387| static ULONG GetSystemReady(void) { return
| SystemReady; }
52388| static NTSTATUS CreateFilesForVolume(
| PDEVICE_OBJECT Volume, PVOID AbortEvent );
52389| static NTSTATUS TearDownCacheForVolume(
| PDEVICE_OBJECT Volume );
52390| static NTSTATUS InitializeFileNamesForVolume (
| PDEVICE_OBJECT Volume );
52391| static NTSTATUS OpenFilesForVolume( PDEVICE_OBJECT
| Volume );
52392| static NTSTATUS GetStateForVolume( PDEVICE_OBJECT
| Volume );
52393| static BOOLEAN CheckIfLoadNeeded( PDEVICE_OBJECT
| Volume );
52394| static NTSTATUS CopyClusterRegistryToLocalRegistry(
| PDEVICE_OBJECT Volume, PUNICODE_STRING LocalReg );
52395| static ULONG QueryResetPsm();
52396| static ULONG QueryNoPsm();
52397| static void DisableResetPsm();
52398| static void DisableNoPsm();
52399|
52400| static ULONG CalculateChecksum ( ULONG
| DataSizeInBytes, const void *DataBuffer );
52401|
52402| friend class
| TreeSearcher_MakeVirtualWritesPersistent;
52403| NTSTATUS MakeVirtualWritesPersistent();
52404|
52405| static NTSTATUS FindVirginSpace (
52406| PDEVICE_OBJECT Volume,
52407| tVirginMap &VirginMap,
52408| ULONG &ClusterSizeInBytes,
52409| LARGE_INTEGER &TotalClusters );
52410|
52411| private:
| //-----
| -----

```

```

52412| static ULONG ClassInitd;
52413| ErrorCode SetVolumeInternal( PDEVICE_OBJECT Volume,
    | pInternalSnapShot MySnapShot, ULONG SnapShotSequence );
52414| BOOLEAN ModelsDefunct( keyType PreviousKey, keyType
    | CurrentKey);
52415| tTreeLeaf *ScanForDeadNode( keyType PreviousKey,
    | ULONG NumNodesToInspect );
52416| void FreeDeadNodes(void);
52417|
52418|
52419| PDEVICE_OBJECT Volume;
52420| PFILTERED_EXTENSION DevExt;
52421| void FreeNodeAndBits( void *Ptr );
52422| class PersistentDictionary *Next;
52423|
52424| pShared Shared;
52425|
52426| protected:
52427| ULONG Flags; //needs to be protected so
    | TemporaryDictionary can access it
52428|
52429| private:
52430| // ULONG SnapShotIndex;
52431| pInternalSnapShot SnapShot;
52432|
52433| static ULONG SystemReady;
52434| static ULONG GhostBusterTime;
52435| static PersistentDictionary *ListHead;
52436|
52437| // When adding/deleting any static variables modify
    | the
52438| // InitClass method to initialize the values
52439| static ULONG MaxNumUserSnapShots;
52440| static ULONG ShareFiles;
52441| static ULONG QPeriod;
52442| static ULONG QTimeout;
52443| static ULONG SectorSize; //
    | FIXFIXFIX: We don't necessarily have same sector size
    | on all volumes!
52444| static ULONG NumSavedHeaderBlocks; //
    | (FIXFIXFIX see above) used for rotated headers
52445| static ULONG NextHeaderBlockToSave; //
    | (FIXFIXFIX see above) used for rotated headers
52446| static signed long Updating;
52447| static ULONG PSMFreeSpaceOptions;
52448| static ULONG NoPsm;
52449| static ULONG ResetPsm;
52450| static ULONG NoPsm_Disabled;
52451| static ULONG ResetPsm_Disabled;
52452|

```

```

52453|
52454| static ErrorCode MakeMountPoints(void);
52455| static ErrorCode GetRegistrySettings(
    | PUNICODE_STRING RegistryPath );
52456| static ErrorCode OpenFiles( ULONG Stage, PVOID
    | AbortEvent );
52457|
52458| static ErrorCode OpenAFile(
52459|     WCHAR      *File,
52460|     HANDLE      &FileHandle,
52461|     PFILE_OBJECT &FileObject,
52462|     HANDLE      &WaitHandle,
52463|     PVOID      &WaitObject );
52464|
52465| static ErrorCode CloseAFile (
52466|     HANDLE      &FileHandle,
52467|     PFILE_OBJECT &FileObject,
52468|     HANDLE      &WaitHandle,
52469|     PVOID      &WaitObject );
52470|
52471| static ErrorCode ReloadStateFromDisk(void);
52472|
52473| static pInternalSnapShot
    | GetNextFreeSnapShot(PDEVICE_OBJECT Volume, ULONG
    | SequenceNumber );
52474|
52475| ULONG GetNextCacheLocation ( ULONG
    | NumberOfGranulesNeeded );
52476| void FreeCacheLocation ( ULONG GranuleIndex );
52477| static BOOLEAN ValidateChecksum ( ULONG
    | DataSizeInBytes, const void *DataBuffer, ULONG
    | StoredChecksum );
52478| static BOOLEAN IndexSectorIsValid ( pIndexSector x,
    | ULONG indexSectorNumber, NTSTATUS &status );
52479|
52480| #if DEBUG_VALIDATE_DIFF_GRANULES
52481| static bool ValidateDiffGranule (
52482|     tIndexSector *,
52483|     void *diffGranuleBuffer,
52484|     PFILTERED_EXTENSION,
52485|     ULONG granuleNumber);
52486| #endif /*DEBUG_VALIDATE_DIFF_GRANULES*/
52487|
52488| static NTSTATUS EraseIndexInfo (
    | PFILTERED_EXTENSION DevExt, ULONG CacheNode );
52489| static NTSTATUS SaveIndexInfo ( PFILTERED_EXTENSION
    | DevExt, ULONG DasdGranule, ULONG SnapshotNumber, ULONG
    | CacheNode, ULONG DataChecksum, ULONG Flags );
52490| static NTSTATUS WriteIndexSector (
    | PFILTERED_EXTENSION DevExt, ULONG GranuleIndex, const

```

```

    | tIndexSector &IndexSector );
52491|
52492|    static void CreditInterveningGranulesInTree
    | (PRTL_BITMAP GranuleBitMap, tTree *Tree, ULONG LowerSS,
    | ULONG UpperSS);
52493| // static NTSTATUS RebuildDictionary (void);
52494|    static void InitHeaderBitmaps(void);
52495|
52496|    static NTSTATUS LoadHeader(PDEVICE_OBJECT Volume);
52497|    static NTSTATUS ReadHeaderBlock (PDEVICE_OBJECT
    | Volume, ULONG HeaderBlockIndex, pHeader
    | &SmallLocalHeader, pHeader &LocalHeader);
52498|
52499|    static NTSTATUS ReadHeader (
52500|        PDEVICE_OBJECT Volume,
52501|        pHeader &MostRecentValidHeader,
52502|        pHeader &SmallMostRecentValidHeader );
52503|
52504|    // Note: After calling ReadHeader, it is caller's
    | responsibility to call FREE_POINTER()
52505|    // on MostRecentValidHeader if it is not NULL, and
    | same with SmallMostRecentValidHeader.
52506|
52507|    static NTSTATUS FreeHeader(PDEVICE_OBJECT Volume);
52508|    static void SaveHeaderThread( void *Psh );
52509|    static pkSnapshotMaster
    | FindMasterSnapShotOnOtherVolume ( pInternalSnapShot
    | SnapShot );
52510|    static skSnapshotMaster *GetAMasterSnapShot( struct
    | _OT_USER_ *User, pInternalSnapShot MySnapShot );
52511|    static struct skSnapshotEntry
    | *GetSnapShotForMaster( skSnapshotMaster
    | *MasterSnapShot, PDEVICE_OBJECT VolumeObject,
    | pInternalSnapShot SnapShot );
52512|    static void DumpSizeOfs(void);
52513|    static void DumpHeader(void);
52514|    static bool BetterThan ( pHeader h1, pHeader h2 );
52515|
52516|    static pInternalSnapShot AllocSnapShot(void);
52517|    static void FreeSnapShot( pInternalSnapShot
    | *SnapShot );
52518|    static void AddSnapShotToList( PLIST_ENTRY Head,
    | pInternalSnapShot SnapShot );
52519|    static void RemoveSnapShotFromList(
    | pInternalSnapShot SnapShot );
52520|    static pInternalSnapShot FindSnapShotFromSequence(
    | PDEVICE_OBJECT Volume, ULONG SequenceNumber );
52521|
52522|    static NTSTATUS AllocMemoryForVolume(
    | PDEVICE_OBJECT Volume, BOOLEAN UseDefaults );

```



```

52523| static NTSTATUS FreeMemoryForVolume( PDEVICE_OBJECT
| Volume );
52524| static NTSTATUS ReleaseASnapShotForVolume(
| PDEVICE_OBJECT Volume, pInternalSnapShot SnapShot );
52525| static NTSTATUS RemoveSnapShotDroppings(
| PDEVICE_OBJECT Volume );
52526| static void DumpSnapShot ( tDiskInternalSnapShot *
| );
52527|
52528| NTSTATUS RemoveVirtualWritesFromTree ( ULONG
| ShouldEraseIndexSectors );
52529| #ifdef DEBUG
52530| void DumpVirtualWriteTree();
52531| #endif
52532|
52533| static NTSTATUS FindVirginSpace_GranuleBitmapPhase(
52534| tVirginMap &VirginMap,
52535| pShared Shared,
52536| ULONG ClustersPerGranule,
52537| ULONG &NumExtentsLost,
52538| LARGE_INTEGER TotalClusters );
52539|
52540| static NTSTATUS FindVirginSpace_SnapshotPhase(
52541| tVirginMap &VirginMap,
52542| tVirginMap &TempMap,
52543| pShared Shared,
52544| ULONG ClustersPerGranule,
52545| ULONG &NumExtentsLost );
52546|
52547| static NTSTATUS FindVirginSpace_VolumeBitmapPhase(
52548| tVirginMap &VirginMap,
52549| tVirginMap &TempMap,
52550| ULONG &NumExtentsLost,
52551| PFILE_OBJECT VolumeObject );
52552| };
52553|
52554|
52555| typedef PersistentDictionary *pPersistentDictionary;
52556|
52557|
52558|
52559| //-----
| -----
52560|
52561| class TemporaryDictionary: public PersistentDictionary
52562| {
52563| public:
52564| TemporaryDictionary();
52565| ~TemporaryDictionary();
52566| //virtual ULONG GetSequenceNumber() const { return

```

```

    | 0xffffffff; }
52567| };
52568|
52569|
52570| typedef TemporaryDictionary *pTemporaryDictionary;
52571|
52572|
52573| //-----
    | -----
52574|
52575| class GenericTreeSearcher
52576| {
52577| public:
52578|     GenericTreeSearcher ( tTree &_tree ):
52579|         tree(_tree)
52580|     {}
52581|
52582|     bool visitEveryNode(); // returns true if search
    | was completed, false if nodeCallback ever returned
    | false
52583|
52584|     virtual bool nodeCallback ( tTreeLeaf *node, int
    | depth ) = 0; // returns false to abort search.
52585|
52586| protected:
52587|     bool recursiveTraversal ( tTreeLeaf *node, int
    | depth );
52588|
52589| private:
52590|     tTree &tree;
52591| };
52592|
52593| //-----
    | -----
52594|
52595|
52596| typedef struct sPerSiblingInfo {
52597|     BOOLEAN AlreadyHandled;
52598| } tPerSiblingInfo,*pPerSiblingInfo;
52599|
52600|
52601| //-----
    | -----
52602|
52603| typedef struct sSaveHeaderParameters {
52604|     PDEVICE_OBJECT Volume;
52605|     ULONG ShouldSignalEvent;
52606|     KEVENT Event;
52607| } tSaveHeaderParameters, *pSaveHeaderParameters;
52608|

```

```

52609| //-----
52610| | -----
52611| typedef struct sCloseProcessSpecificHandles {
52612|     HANDLE &Handle;
52613|     PVOID &Object;
52614|     KEVENT Event;
52615|
52616|     sCloseProcessSpecificHandles ( HANDLE &_Handle,
52617|         | PVOID &_Object ):
52618|         Handle(_Handle),
52619|         Object(_Object)
52620|     {
52621|         RtlZeroMemory(&Event,sizeof(Event));
52622|     } *pCloseProcessSpecificHandles;
52623|
52624|
52625| void CloseProcessSpecificHandles(
52626|     | pCloseProcessSpecificHandles Psh );
52627| extern PEPROCESS GlobalSystemProcessId;
52628| void CloseProcessHandle ( HANDLE &h, PVOID &o );
52629|
52630| NTSTATUS Rebuild_DeleteJunctionsOnVolume (
52631|     | PDEVICE_OBJECT Volume );
52632| /*--- end of file perapi.h ---*/
52633|
52634|
52635|
52636| File Listing: perdict.cpp
52637|
52638| #include "precomp.h"
52639| #include "buildnum.h"
52640|
52641| #define INVALID_BIT_INDEX ((ULONG)(-1))
52642|
52643| // when in debugger, set this to 1 to not rebuild the
52644|     | dictionary during boot up
52645| ULONG DebugReset=0;
52646| //ULONGLONG
52647|     | PersistentDictionary::MostRecentSnapshotTime=0;
52648| //PersistentDictionary::pInternalSnapShot
52649|     | PersistentDictionary::MostRecentSnapshot=NULL;
52650| ULONG
52651|     | PersistentDictionary::GhostBusterTime=FALSE;
52652| ULONG     PersistentDictionary::QPeriod=0;
52653| ULONG     PersistentDictionary::QTimeout=0;

```

```

52651| ULONG          PersistentDictionary::ShareFiles=0;
52652| pPersistentDictionary PersistentDictionary::ListHead =
    | NULL;
52653| ULONG          PersistentDictionary::SectorSize=0;
52654| ULONG
    | PersistentDictionary::NumSavedHeaderBlocks =
    | DEFAULT_HEADER_BLOCKS_TO_SAVE;
52655| ULONG
    | PersistentDictionary::NextHeaderBlockToSave = 0;
52656| signed long     PersistentDictionary::Updating=0;
52657| ULONG
    | PersistentDictionary::MaxNumUserSnapShots =
    | MAX_USER_SNAPSHOTS;
52658| ULONG
    | PersistentDictionary::PSMFreeSpaceOptions = 0;
52659| ULONG          PersistentDictionary::ClassInited = 0;
52660| ULONG          PersistentDictionary::SystemReady= 0;
52661|
52662| ULONG PersistentDictionary::NoPsm = FALSE;
52663| ULONG PersistentDictionary::ResetPsm = FALSE;
52664|
52665| ULONG PersistentDictionary::NoPsm_Disabled = FALSE;
52666| ULONG PersistentDictionary::ResetPsm_Disabled = FALSE;
52667|
52668| #define CHECKSUM_IGNORE_DWORD 0xf5e4d3c1
52669|
52670| ULONG
52671| PersistentDictionary::CalculateChecksum (
52672|          ULONG
    | DataSizeInBytes,
52673|          const void
    | *DataBuffer )
52674| {
52675|     ULONG sum = 0xc9d4b5a2;
52676|     ULONG multiplier = 0x1a2b3c4d;
52677|     const unsigned char *p = (const unsigned char *)
    | DataBuffer;
52678|     ULONG i;
52679|
52680|     for ( i=0; i < DataSizeInBytes; ++i ) {
52681|         sum ^= (0x100 + *p++) * multiplier++;
52682|         sum = (sum << 9) | (sum >> (32-9));    //
    | rotate left 9 bits
52683|     }
52684|
52685|     if ( sum == CHECKSUM_IGNORE_DWORD ) {
52686|         sum = 0xffffffff; // not allowed to be
    | CHECKSUM_IGNORE_DWORD
52687|     }
52688|

```

```

52689|    return sum;
52690| }
52691|
52692| BOOLEAN
52693| PersistentDictionary::ValidateChecksum (
52694|     ULONG
52695|     | DataSizeInBytes,
52696|     | *DataBuffer,
52697|     | StoredChecksum )
52698| {
52699|     BOOLEAN isValid = TRUE;
52700|     // If the stored checksum is CHECKSUM_IGNORE_DWORD,
52701|     | it means we should ignore it.
52702|     if ( StoredChecksum != CHECKSUM_IGNORE_DWORD ) {
52703|         ULONG CalcChecksum = CalculateChecksum
52704|         | (DataSizeInBytes, DataBuffer);
52705|         isValid = (CalcChecksum == StoredChecksum);
52706|     }
52707|     return isValid;
52708| }
52709|
52710| //-----
52711| | -----
52712| // Stuff for index file:
52713|
52714| BOOLEAN MemCompareToByte( const void *Buffer, unsigned
52715|     | char Byte, ULONG Size )
52716| {
52717|     const ULONG *p = (const ULONG*)Buffer;
52718|     const ULONG NumDWords = Size / sizeof(*p);
52719|     const ULONG Pattern = (ULONG)Byte * 0x01010101;
52720|
52721|     ASSERT(Size % sizeof(ULONG)==0);
52722|     for ( ULONG i=0; i<NumDWords; ++i ) {
52723|         if ( *p++ != Pattern ) {
52724|             return FALSE;
52725|         }
52726|     }
52727|     return TRUE;
52728| }
52729| //-----
52730| | -----

```

```

52731| BOOLEAN
52732| PersistentDictionary::IndexSectorIsValid (
52733|                                     pIndexSector
52734|     | x,
52735|                                     ULONG
52736|     | indexSectorNumber,
52737|                                     NTSTATUS
52738|     | &status )
52739| {
52740|     BOOLEAN isValid=FALSE;
52741|     BOOLEAN isEmpty =
52742|         | MemCompareToByte(x,0x00,SectorSize);
52743|     status = STATUS_SUCCESS;
52744|     if ( !isEmpty ) {
52745|         isValid = ValidateChecksum (
52746|             SectorSize -
52747|             | sizeof(ULONG),
52748|             | &(x->Filler[sizeof(ULONG)]),
52749|             | x->Info.Prefix.IndexChecksum );
52750|         if ( isValid ) {
52751|             isValid = (x->Info.DiskNode[0].CacheNode ==
52752|                 | indexSectorNumber);
52753|             if ( isValid ) {
52754|                 isValid = (x->Info.Prefix.PrefixSize ==
52755|                     | sizeof(tIndexSectorPrefix));
52756|                 if ( isValid ) {
52757|                     isValid =
52758|                         | (x->Info.DiskNode[0].SnapshotNumber > 0);
52759|                 } else {
52760|                     status = STATUS_FILE_CORRUPT_ERROR;
52761|                 }
52762|             } else {
52763|                 status = STATUS_FILE_CORRUPT_ERROR;
52764|             }
52765|         } else {
52766|             status = STATUS_FILE_CORRUPT_ERROR;
52767|         }
52768|     } else {
52769|         return isValid;
52770|     }
52771| }
52772| //-----

```

```

| -----
52771|
52772| NTSTATUS PersistentDictionary::EraseIndexInfo (
52773|     PFILTERED_EXTENSION    DevExt,
52774|     ULONG                    CacheNode )
52775| {
52776|     NTSTATUS status = 0;
52777|     Debug(DEBUG_DICT,("pd::EraseIndexInfo (DevExt=%08x,
|     CacheNode=%08x)\n",DevExt,CacheNode));
52778|
52779|     tIndexSector IndexSector = {0};
52780|     IO_STATUS_BLOCK IoStatus;
52781|     LARGE_INTEGER Location;
52782|     Location.QuadPart = SectorSize*(unsigned
|     __int64)CacheNode;
52783|
52784|     status = PsmWriteToFile(
52785|         &DevExt->Cache.IndexFile,
52786|         &IoStatus,
52787|         (PVOID)&IndexSector,
52788|         SectorSize,
52789|         &Location,
52790|         DevExt->DoDirectIO,
52791|         &DevExt->Cache.DirectAccessResource );
52792|
52793|     if ( !NT_SUCCESS(status) ) {
52794|         Debug(DEBUG_DICT,( "!!! Error 0x%08lx writing
|         to index file.\n", (unsigned long)status ));
52795|         ASSERT(FALSE);
52796|     }
52797|
52798|     return status;
52799| }
52800|
52801| //-----
| -----
| -----
52802|
52803| NTSTATUS PersistentDictionary::WriteIndexSector (
52804|     PFILTERED_EXTENSION    DevExt,
52805|     ULONG                    GranuleIndex,
52806|     const tIndexSector      &IndexSector )
52807| {
52808|     NTSTATUS status = 0;
52809|     IO_STATUS_BLOCK IoStatus;
52810|     LARGE_INTEGER Location;
52811|     ULONG BytesToWrite = SectorSize;
52812|
52813|     __try {
52814|         Location.QuadPart = SectorSize * ((unsigned

```

```

    | __int64)GranuleIndex);
52815|     status = PsmWriteToFile(
52816|         &DevExt->Cache.IndexFile,
52817|         &IoStatus,
52818|         &IndexSector,
52819|         SectorSize,
52820|         &Location,
52821|         DevExt->DoDirectIO,
52822|         &DevExt->Cache.DirectAccessResource );
52823| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
52824|     status = GetExceptionCode();
52825|     Debug(DEBUG_DICT,("Exception %08x in
    | pd::WriteIndexSector\n",status));
52826| }
52827|
52828| if ( !NT_SUCCESS(status) ) {
52829|     Debug(DEBUG_DICT,( "!!! Error 0x%08lx writing
    | to index file.\n", (unsigned long)status ));
52830|     ASSERT(FALSE);
52831| }
52832|
52833| return status;
52834| }
52835|
52836| //-----
    | -----
    | -----
52837|
52838|
52839| NTSTATUS PersistentDictionary::SaveIndexInfo (
52840|     PFILTERED_EXTENSION    DevExt,
52841|     ULONG                   DasdGranule,
52842|     ULONG                   SnapshotNumber,
52843|     ULONG                   CacheNode,
52844|     ULONG                   DataChecksum,
52845|     ULONG                   Flags )
52846| {
52847|     Profile("pd::SaveIndexInfo");
52848|
52849|     ULONG i = 0;
52850|
52851|     Debug(DEBUG_DICT,( "SaveIndexInfo (DevExt=%08x,
    | DasdGran=%08x, SS=%08x, CacheNode=%08x,
    | DataChecksum=0x%08lx, Flags=%08x)\n",
52852|         DevExt,
52853|         (unsigned long) DasdGranule,
52854|         (unsigned long) SnapshotNumber,
52855|         (unsigned long) CacheNode,
52856|         (unsigned long) DataChecksum,

```



```

52857|     Flags));
52858|
52859|     ASSERT (SectorSize == sizeof(tIndexSector));
52860|
52861|     tIndexSector IndexSector;
52862| #if 0
52863|     // Scoot the disk nodes "down", deleting the last
52864|     | one, so that we can stick
52865|     // new one in the first slot.
52866|     for ( i = DISK_NODES_PER_INDEX_SECTOR - 1; i>0; --i
52867|     | ) {
52868|         IndexSector.Info.DiskNode[i] =
52869|         | IndexSector.Info.DiskNode[i-1];
52870|     }
52871| #endif
52872|
52873|     // Now put the new disk node data in the first
52874|     | slot...
52875|     IndexSector.Info.DiskNode[0].DasdGranule      =
52876|     | DasdGranule;
52877|     IndexSector.Info.DiskNode[0].SnapshotNumber    =
52878|     | SnapshotNumber;
52879|     IndexSector.Info.DiskNode[0].CacheNode         =
52880|     | CacheNode;
52881|     IndexSector.Info.DiskNode[0].DataChecksum      =
52882|     | DataChecksum;
52883|     IndexSector.Info.DiskNode[0].Volumeld          =
52884|     | DevExt->Volumeld;
52885|     IndexSector.Info.DiskNode[0].Flags             =
52886|     | Flags;
52887|
52888|     // Fill in all prefix information ***EXCEPT*** for
52889|     | checksum (done after everything else):
52890|
52891|     IndexSector.Info.Prefix.PrefixSize =
52892|     | sizeof(tIndexSectorPrefix);
52893|     IndexSector.Info.Prefix.Version    =
52894|     | PSM_INDEX_CURRENT_VERSION;
52895|     IndexSector.Info.Prefix.Sequence   =
52896|     | (DevExt->NextIndexSequenceNumber)++;
52897|     KeQuerySystemTime
52898|     | (&(IndexSector.Info.Prefix.DateWritten));
52899|
52900|     // The checksum calculation must be done last, so
52901|     | that remaining data is valid:
52902|
52903|     IndexSector.Info.Prefix.IndexChecksum =
52904|     | CalculateChecksum (

```

```

52890|     SectorSize - sizeof(ULONG),
52891|     &IndexSector.Filler[sizeof(ULONG)] );
52892|
52893|     NTSTATUS status = WriteIndexSector (DevExt,
    | CacheNode, IndexSector);
52894|     return status;
52895| }
52896|
52897| //-----
    | -----
52898|
52899| PersistentDictionary::pInternalSnapShot
    | PersistentDictionary::AllocSnapShot()
52900| {
52901|     pInternalSnapShot
    | p=(PersistentDictionary::pInternalSnapShot)MemAllocatePo
    | olWithTag(PagedPool,sizeof(tlInternalSnapShot),PSM_INTERN
    | AL_SNAPSHOT);
52902|     if ( p ) {
52903|         Debug(DEBUG_DICT,("pd::AllocSnapShot: Allocated
    | snapshot %08x\n",p));
52904|         RtlZeroMemory(p,sizeof(tlInternalSnapShot));
52905|     }
52906|     return p;
52907| }
52908|
52909| //-----
    | -----
52910|
52911| void PersistentDictionary::FreeSnapShot(
    | PersistentDictionary::pInternalSnapShot *SnapShot )
52912| {
52913|     Debug(DEBUG_DICT,("pd::FreeSnapShot: Freeing
    | snapshot %08x\n",SnapShot));
52914|     RtlZeroMemory(*SnapShot,sizeof(tlInternalSnapShot));
52915|     MemFreePool(*SnapShot);
52916|     *SnapShot = NULL;
52917| }
52918|
52919| //-----
    | -----
52920|
52921| void PersistentDictionary::AddSnapShotToList(
    | PLIST_ENTRY Head,
    | PersistentDictionary::pInternalSnapShot SnapShot )
52922| {
52923|     Debug(DEBUG_DICT,("pd::AddSnapShotToList: Adding
    | %08x to snapshot list\n",SnapShot));
52924|     InsertTailList(Head,&SnapShot->ListEntry);
52925| }

```

```

52926|
52927| //-----
    | -----
52928|
52929| void PersistentDictionary::RemoveSnapShotFromList(
    | PersistentDictionary::pInternalSnapShot SnapShot )
52930| {
52931|     Debug(DEBUG_DICT,("pd::RemoveSnapShotFromList:
    | Removing %08x from snapshot list\n",SnapShot));
52932|     RemoveEntryList(&SnapShot->ListEntry);
52933| }
52934|
52935| //-----
    | -----
52936|
52937| PersistentDictionary::pInternalSnapShot
    | PersistentDictionary::FindSnapShotFromSequence (
52938|     PDEVICE_OBJECT    Volume,
52939|     ULONG              SequenceNumber )
52940| {
52941|     PLIST_ENTRY p;
52942|     pInternalSnapShot s=NULL;
52943|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
52944|
52945|     p=DevExt->Cache.SnapShotHead.Flink;
52946|
52947|     while ( p!=&DevExt->Cache.SnapShotHead ) {
52948|
    | s=CONTAINING_RECORD(p,tInternalSnapShot,ListEntry);
52949|
52950|         if ( s->Permanent.SequenceNumber ==
    | SequenceNumber ) {
52951|             break;
52952|         }
52953|         p=p->Flink;
52954|         s = NULL;
52955|     }
52956|
52957|     //Debug(DEBUG_DICT,("pd::FindSnapShotFromSequence
    | (Volume=%08x, Sequence=%08x) returning
    | %08x\n",Volume,SequenceNumber,s));
52958|     return s;
52959| }
52960|
52961| //-----
    | -----
52962|
52963| ULONG
    | PersistentDictionary::IsCacheSnapShotCreationThresholdRe

```

```

    | ached ()
52964| {
52965|     ULONG CacheWarningThreshold = (ULONG)(((unsigned
    | __int64)DevExt->Cache.PSManBitMapMaxSize *
    | DevExt->Cache.CacheWarningThresholdPercent) / 100);
52966|     ULONG CacheSnapDeleteThreshold = (ULONG)(((unsigned
    | __int64)DevExt->Cache.PSManBitMapMaxSize *
    | DevExt->Cache.CacheFullThresholdPercent) / 100);
52967|
52968|     signed long GranuleSpan = ((signed
    | long)CacheSnapDeleteThreshold - (signed
    | long)CacheWarningThreshold)/5; //20 percent of the
    | gap
52969|     static const signed long MIN_GRANULE_SPAN =
    | 8*1024*1024/GRANULE_SIZE; //8MB of space between
    | stopping new ss create and deleting old ss
52970|     if ( GranuleSpan < MIN_GRANULE_SPAN ) {
52971|         GranuleSpan = MIN_GRANULE_SPAN;
52972|
    | Debug(DEBUG_DICT,("pd::IsCacheSnapShotCreationThresholdR
    | eached: Increased GranuleSpan to minimum
    | %08x\n",GranuleSpan));
52973|     }
52974|
52975|     ASSERT(GranuleSpan < (signed
    | long)CacheSnapDeleteThreshold);
52976|     ULONG CacheSnapCreationThreshold =
    | CacheSnapDeleteThreshold - GranuleSpan;
52977|     if (
    | CacheSnapCreationThreshold<CacheWarningThreshold &&
    | CacheWarningThreshold<CacheSnapDeleteThreshold ) {
52978|         CacheSnapCreationThreshold =
    | CacheWarningThreshold;
52979|
    | Debug(DEBUG_DICT,("pd::IsCacheSnapShotCreationThresholdR
    | eached: Moved creation threshold to warning
    | threshold\n"));
52980|     }
52981|
52982|
    | Debug(DEBUG_DICT,("pd::IsCacheSnapShotCreationThresholdR
    | eached:\n"));
52983|     Debug(DEBUG_DICT,("    CacheWarningThreshold
    | = %08x\n", CacheWarningThreshold));
52984|     Debug(DEBUG_DICT,("    CacheSnapDeleteThreshold
    | = %08x\n", CacheSnapDeleteThreshold));
52985|     Debug(DEBUG_DICT,("    GranuleSpan
    | = %08x\n", GranuleSpan));
52986|     Debug(DEBUG_DICT,("
    | CacheSnapCreationThreshold = %08x\n",

```

```

    | CacheSnapCreationThreshold));
52987|   Debug(DEBUG_DICT,("    Current Cache Size
    | = %08x\n", DevExt->Cache.CurrentCacheFileSize));
52988|
52989|   ULONG ReturnValue = 0; // indicates that creating
    | a snapshot is OK
52990|   if ( DevExt->Cache.CurrentCacheFileSize >
    | CacheSnapCreationThreshold ) {
52991|       Debug(DEBUG_DICT,("    *** New snapshot
    | creation will be DENIED!\n"));
52992|       ReturnValue = CacheSnapCreationThreshold; //
    | indicates that creating a snapshot is NOT ALLOWED
52993|   } else {
52994|       Debug(DEBUG_DICT,("    --- New snapshot
    | creation will be allowed.\n"));
52995|   }
52996|
52997|   return ReturnValue;
52998| }
52999|
53000| //-----
    | -----
53001|
53002| ULONG
    | PersistentDictionary::IsCacheWarningThresholdReached ()
53003| {
53004|   ULONG CacheWarningThreshold = (ULONG)((((unsigned
    | __int64)DevExt->Cache.PSManBitMapMaxSize *
    | DevExt->Cache.CacheWarningThresholdPercent) / 100);
53005|   if (
    | DevExt->Cache.CurrentCacheFileSize>CacheWarningThreshold
    | ) {
53006|       return CacheWarningThreshold;
53007|   } else {
53008|       return 0;
53009|   }
53010| }
53011|
53012| //-----
    | -----
53013|
53014| PersistentDictionary::pInternalSnapShot
    | PersistentDictionary::GetNextFreeSnapShot (
53015|   PDEVICE_OBJECT Volume,
53016|   ULONG SequenceNumber )
53017| {
53018|   PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
53019|
53020|   Debug(DEBUG_DICT,("GetNextFreeSnapShot: called -

```

```

    | Volume=%08x\n",Volume));
53021|   pInternalSnapShot p=AllocSnapShot();
53022|   if ( p ) {
53023|       ASSERT (DevExt->Cache.Header != NULL);
53024|       ASSERT (SequenceNumber >
    | DevExt->Cache.Header->HighestSnapNumber);
53025|       p->Permanent.SequenceNumber =
    | DevExt->Cache.Header->HighestSnapNumber =
    | SequenceNumber;
53026|
    | AddSnapShotToList(&DevExt->Cache.SnapShotHead,p);
53027|       p->ReferenceCount++;
53028|   }
53029|   Debug(DEBUG_DICT,("GetNextFreeSnapShot: returning
    | %08x\n",p));
53030|   return p;
53031| }
53032|
53033| //-----
    | -----
53034| // Returns a valid position or INVALID_POSITION_VALUE
    | if no positions available
53035|
53036| ULONG PersistentDictionary::GetNextCacheLocation (
    | ULONG GranulesWeNeed )
53037| {
53038|   ULONG BitIndex = INVALID_BIT_INDEX;
53039|   Profile("pd::GetNextCacheLocation");
53040|   ASSERT(DevExt != NULL);
53041|
53042|   // we only support 31 bits of cache file space
53043|   ASSERT(DevExt->Cache.PSManBitMapSize<=0x7ffffff);
53044|
    | ASSERT(DevExt->Cache.PSManBitMapMaxSize<=0x7ffffff);
53045|
53046|   Debug(DEBUG_DICT,("pd::GetNextCacheLocation - about
    | to acquire PSManBitMapMutex\n"));
53047|   pmAcquireMutex ( &DevExt->Cache.PSManBitMapMutex,
    | NULL );
53048|   __try {
53049|       if ( DevExt->Cache.PSManBitHint +
    | GranulesWeNeed > DevExt->Cache.PSManBitMapSize ) {
53050|           // This hint can't possibly work, so start
    | over.
53051|           DevExt->Cache.PSManBitHint = 0;
53052|       }
53053|
53054|       PsmBitPositionValidate
    | (DevExt->Cache.PSManBitMapBuffer,
    | DevExt->Cache.PSManBitHint);

```

```

53055|     BitIndex =
| RtlFindClearBitsAndSet(DevExt->Cache.PSManBitMapBuffer,G
| ranulesWeNeed,DevExt->Cache.PSManBitHint);
53056|     if ( BitIndex==INVALID_BIT_INDEX &&
| DevExt->Cache.PSManBitHint>0 ) {
53057|         // try again from the beginning of the bit
| array
53058|         DevExt->Cache.PSManBitHint = 0;
53059|         BitIndex =
| RtlFindClearBitsAndSet(DevExt->Cache.PSManBitMapBuffer,G
| ranulesWeNeed,DevExt->Cache.PSManBitHint);
53060|     }
53061|
53062|     if ( BitIndex == INVALID_BIT_INDEX ) {
53063|         // INVALID_BIT_INDEX == 32 bits, we can
| only return 31 bits
53064|         BitIndex = INVALID_POSITION_VALUE;
53065|     } else {
53066|         // We found the bits we needed. Now update
| the hint for the next time.
53067|         DevExt->Cache.PSManBitHint = BitIndex +
| GranulesWeNeed;
53068|         DevExt->Cache.CurrentCacheFileSize +=
| GranulesWeNeed;
53069|
53070|         if ( DevExt->Cache.CurrentCacheFileSize >
| DevExt->Cache.PSManBitMapSize ) {
53071|             DevExt->Cache.PSManBitMapSize =
| DevExt->Cache.CurrentCacheFileSize;
53072|         }
53073|     }
53074| } __finally {
53075|     pmReleaseMutex (
| &DevExt->Cache.PSManBitMapMutex );
53076|     Debug(DEBUG_DICT,("pd::GetNextCacheLocation -
| just released PSManBitMapMutex\n"));
53077| }
53078|
53079| ASSERT(BitIndex<=0x7fffffff);
53080| return BitIndex;
53081| }
53082|
53083| //-----
| -----
53084|
53085| void PersistentDictionary::FreeCacheLocation ( ULONG
| GranuleIndex )
53086| {
53087|     Profile("pd::FreeCacheLocation");
53088|

```

```

53089|  #if DEBUG_INDEX_EOF
53090|      Debug(DEBUG_DICT,("pd::FreeCacheLocation:
    | GranuleIndex=%08x, DevExt=%08x - about to acquire
    | PsManBitMapMutex\n",GranuleIndex,DevExt));
53091|  #endif /*DEBUG_INDEX_EOF*/
53092|
53093|  pmAcquireMutex ( &DevExt->Cache.PSManBitMapMutex,
    | NULL );
53094|  __try {
53095|      | PsmBitPositionValidate(DevExt->Cache.PSManBitMapBuffer,
    | GranuleIndex);
53096|      ASSERT (
    | RtlCheckBit(DevExt->Cache.PSManBitMapBuffer,GranuleIndex
    | ) != 0 );
53097|      RtlClearBits ( DevExt->Cache.PSManBitMapBuffer,
    | GranuleIndex, 1 );
53098|      ASSERT ( DevExt->Cache.CurrentCacheFileSize > 0
    | );
53099|      InterlockedDecrement(
    | (PLONG)&DevExt->Cache.CurrentCacheFileSize );
53100|  } __finally {
53101|      pmReleaseMutex (
    | &DevExt->Cache.PSManBitMapMutex );
53102|  }
53103| }
53104|
53105|
53106| //-----
    | -----
    | -----
53107| // translate values in header.psm to PSM_SS_BIT_xxx
    | combinations ...
53108|
53109| unsigned char PSM_SS_HeaderToFlags ( unsigned char
    | HeaderValue )
53110| {
53111|     unsigned char SnapShotFlags =
    | PSM_SS_FLAG_P_READONLY;
53112|
53113|     switch ( HeaderValue ) {
53114|         case PSM_SS_FLAG_HEADER_READWRITE:
53115|             SnapShotFlags = PSM_SS_FLAG_P_READWRITE;
53116|             break;
53117|
53118|         case PSM_SS_FLAG_HEADER_READWRITE_PERSISTENT:
53119|             SnapShotFlags =
    | PSM_SS_FLAG_P_READWRITE_PVW;
53120|             break;
53121|

```



```

53122|     case PSM_SS_FLAG_HEADER_READONLY:
53123|         SnapShotFlags = PSM_SS_FLAG_P_READONLY;
53124|         break;
53125|
53126|     default:
53127|         Debug(DEBUG_DICT,("!!!
    | PSM_SS_HeaderToFlags: Unknown
    | HeaderValue=%02x\n",HeaderValue));
53128|         ASSERT(FALSE);
53129|     }
53130|
53131|     return SnapShotFlags;
53132| }
53133|
53134|
53135| //-----
    | -----
    | -----
53136| // translate PSM_SS_BIT_xxx combinations to values in
    | header.psm ...
53137|
53138| unsigned char PSM_SS_FlagsToHeader ( unsigned char
    | SnapShotFlags )
53139| {
53140|     unsigned char HeaderValue =
    | PSM_SS_FLAG_HEADER_READONLY;
53141|
53142|     // We should never find temporary snapshot stuff in
    | the header...
53143|     ASSERT ( PSM_SS_IsPersistent(SnapShotFlags) );
53144|     ASSERT ( !(SnapShotFlags &
    | PSM_SS_BIT_SAVE_TEMP_ON_EXIT) );
53145|
53146|     if ( PSM_SS_IsReadWrite(SnapShotFlags) ) {
53147|         // Read/Write snapshot
53148|         if ( SnapShotFlags &
    | PSM_SS_BIT_VIRTUAL_WRITES_PERSISTENT ) {
53149|             HeaderValue =
    | PSM_SS_FLAG_HEADER_READWRITE_PERSISTENT;
53150|         } else {
53151|             HeaderValue = PSM_SS_FLAG_HEADER_READWRITE;
53152|         }
53153|
53154|     } else {
53155|         // Read-Only snapshot
53156|
53157|         // Cannot have persistent virtual writes on a
    | read-only snapshot!
53158|         ASSERT ( !(SnapShotFlags &
    | PSM_SS_BIT_VIRTUAL_WRITES_PERSISTENT) );

```

```

53159|    }
53160|
53161|    return HeaderValue;
53162| }
53163|
53164| //-----
    | -----
    | -----
53165|
53166| void PersistentDictionary::DumpSnapShot (
    | PersistentDictionary::tDiskInternalSnapShot *SnapShot )
53167| {
53168| #ifdef DEBUG
53169|     ASSERT (SnapShot != NULL);
53170|     if ( SnapShot ) {
53171|         Debug(DEBUG_DICT,("Dump of
    | tDiskInternalSnapShot %08x\n",SnapShot));
53172|         Debug(DEBUG_DICT,("  SequenceNumber  =
    | %08x\n",SnapShot->SequenceNumber));
53173|         Debug(DEBUG_DICT,("  SnapShotTime    =
    | %016l64x\n",SnapShot->SnapShotTime.QuadPart));
53174|         Debug(DEBUG_DICT,("  ExternalInstance =
    | %08x\n",SnapShot->ExternalInstance));
53175|         Debug(DEBUG_DICT,("  GroupNumber     =
    | %08x\n",SnapShot->GroupNumber));
53176|         Debug(DEBUG_DICT,("  Status          =
    | %08x\n",SnapShot->Status));
53177|         Debug(DEBUG_DICT,("  NumToKeep       =
    | %08x\n",SnapShot->NumToKeep));
53178|         Debug(DEBUG_DICT,("  Priority        =
    | %02x\n",SnapShot->Priority));
53179|         Debug(DEBUG_DICT,("  SnapShotFlags   =
    | %02x\n",SnapShot->SnapShotFlags));
53180|         Debug(DEBUG_DICT,("  Reserved1      =
    | %02x\n",SnapShot->Reserved1));
53181|         Debug(DEBUG_DICT,("  Reserved2      =
    | %02x\n",SnapShot->Reserved2));
53182|         Debug(DEBUG_DICT,("  UserSnapShotName =
    | '%S'\n",SnapShot->UserSnapShotName));
53183|     }
53184| #endif /*DEBUG*/
53185| }
53186|
53187| //-----
    | -----
    | -----
53188|
53189| void PersistentDictionary::SaveHeaderThread( void
    | *Context )
53190| {

```

```

53191| Profile("pd::SaveHeaderThread");
53192|
53193| pSaveHeaderParameters Psh =
    | pSaveHeaderParameters(Context);
53194| GetSnapShotForRead();
53195| ULONG SnapShotAcquired = TRUE;
53196| __try {
53197|     __try {
53198|         ULONG NumberOfPersistentSnapShots=0;
53199|         ULONG TotalNumberOfSnapShots=0;
53200|         LARGE_INTEGER Location={0};
53201|         NTSTATUS Status=0;
53202|         IO_STATUS_BLOCK IoStatus={0};
53203|         PDEVICE_OBJECT Volume = Psh->Volume;
53204|         PFILTERED_EXTENSION DevExt =
            | GetFilteredExtension(Volume);
53205|
53206|         Debug(DEBUG_DICT,("pd::SaveHeaderThread:
            | Volume=%08x, DevExt=%08x\n",Volume,DevExt));
53207|
53208|         // get number of snapshots so we now how
            | much memory to alloc
53209|         PLIST_ENTRY
            | p=DevExt->Cache.SnapShotHead.Flink;
53210|         while ( p!=&DevExt->Cache.SnapShotHead ) {
53211|             ++TotalNumberOfSnapShots;
53212|             pInternalSnapShot s =
                | CONTAINING_RECORD(p,tInternalSnapShot,ListEntry);
53213|             if ( PSM_SS_IsPersistent
                | (s->Permanent.SnapShotFlags) ) {
53214|                 // Only count persistent snapshots,
                | because those are the only ones we will save.
53215|                 ++NumberOfPersistentSnapShots;
53216|             }
53217|             p=p->Flink;
53218|         }
53219|
53220|         ASSERT ( NumberOfPersistentSnapShots <=
            | MAX_SNAPSHOTS_PER_HEADER_FILE );
53221|
53222|         Debug(DEBUG_DICT,("pd::SaveHeaderThread:
            | %08x - %d persistent snapshots active, %d
            | total\n",&DevExt->Cache.SnapShotHead,NumberOfPersistentS
            | napShots,TotalNumberOfSnapShots));
53223|         if ( DevExt->Cache.Header ) {
53224|             // Sizes of the stuff we want to save
53225|             ULONG
                | SizeInMemory=sizeof(tHeader)+(NumberOfPersistentSnapShot
                | s*sizeof(tDiskInternalSnapShot))+sizeof(DWORD);
53226|             ULONG SizeOnDisk =

```

```

    | ROUND_UP(SizeInMemory,DevExt->BytesPerSector);
53227|         ASSERT(SizeOnDisk>=SizeInMemory);
53228|         ASSERT(SizeOnDisk %
    | DevExt->BytesPerSector == 0);
53229|
53230|         // Maximum sizes of header blocks
53231|         ULONG BlockSizeInMemory =
    | sizeof(tHeader)+(MAX_SNAPSHOTS_PER_HEADER_FILE*sizeof(tD
    | iskInternalSnapShot))+sizeof(DWORD);
53232|         ULONG BlockSizeOnDisk =
    | ROUND_UP(BlockSizeInMemory,DevExt->BytesPerSector);
53233|
    | ASSERT(BlockSizeOnDisk>=BlockSizeInMemory);
53234|         ASSERT(BlockSizeOnDisk %
    | DevExt->BytesPerSector == 0);
53235|
53236|         pHeader LocalHeader =
    | (pHeader)MemAllocatePoolWithTag(PagedPool,SizeOnDisk,PSM
    | _DICT_HEADER_TAG);
53237|         if ( LocalHeader ) {
53238|
    | RtlZeroMemory(LocalHeader,SizeOnDisk);
53239|
53240|         // Rotate the wheel! This helps us
    | know if we need to reload index info on cluster
    | failover...
53241|
    | ++(DevExt->Cache.Header->IndexLoadWheel);
53242|
    | Debug(DEBUG_DICT,("pd::SaveHeaderThread:
    | IndexLoadWheel=%08x,
    | DevExt=%08x\n",DevExt->Cache.Header->IndexLoadWheel,DevE
    | xt));
53243|
53244|         // copy the normal header stuff
53245|
    | RtlCopyMemory(LocalHeader,DevExt->Cache.Header,sizeof(tH
    | eader));
53246|
53247|         // move snapshots into new header
53248|         PLIST_ENTRY
    | p=DevExt->Cache.SnapShotHead.Flink;
53249|         ULONG i=0;
53250|         while (
    | p!=&DevExt->Cache.SnapShotHead ) {
53251|             pInternalSnapShot s =
    | CONTAINING_RECORD(p,tInternalSnapShot,ListEntry);
53252|             if ( PSM_SS_IsPersistent
    | (s->Permanent.SnapShotFlags) ) {
53253|

```

```

    | DumpSnapShot(&s->Permanent);
53254|
    | Debug(DEBUG_DICT,("pd::SaveHeaderThread: Adding
    | snapshot seq %08x (status=%08x) to header at pos
    | %08x\n",s->Permanent.SequenceNumber,s->Permanent.Status,
    | i));
53255|             ASSERT ( i <
    | NumberOfPersistentSnapShots );
53256|
    | RtlCopyMemory(&LocalHeader->SnapShots[i],&s->Permanent,s
    | izeof(tDiskInternalSnapShot));
53257|
    | LocalHeader->SnapShots[i].SnapShotFlags =
    | PSM_SS_FlagsToHeader
    | (LocalHeader->SnapShots[i].SnapShotFlags);
53258|
    | LocalHeader->SnapShots[i].Status = s->SnapShotMaster ?
    | s->SnapShotMaster->Status : PSM_PENDING;
53259|             i++;
53260|         } else {
53261|
    | Debug(DEBUG_DICT,("pd::SaveHeaderThread: Excluding
    | temporary snapshot seq
    | %08x\n",s->Permanent.SequenceNumber));
53262|         }
53263|         p=p->Flink;
53264|     }
53265|
53266|
    | ASSERT(i==NumberOfPersistentSnapShots);
53267|
53268|         KeQuerySystemTime (
    | &(LocalHeader->DateTimeWritten) );
53269|         static LARGE_INTEGER LatestSaveTime
    | = {0};
53270|         if (
    | LocalHeader->DateTimeWritten.QuadPart >
    | LatestSaveTime.QuadPart ) {
53271|             LatestSaveTime =
    | LocalHeader->DateTimeWritten;
53272|         } else {
53273|             // Weird problem: this can
    | happen if we are saving header to a RAID
53274|             // with write-through RAM on
    | board. The time between consecutive SaveHeader calls
53275|             // can leave us thinking two
    | headers have been written at the same time, thus
53276|             // during reboot we cannot tell
    | which one is most recent. This may be the
53277|             // cause of our elusive

```

```

    | junction point problems.
53278|          // Fix: If this time seems to
    | be at or before the previous time we saved,
53279|          // then use the previous time
    | plus one.
53280|          // This forces all DateTimes
    | written to be in strictly increasing order.
53281|
53282|          | LocalHeader->DateTimeWritten.QuadPart =
    | ++LatestSaveTime.QuadPart;
53283|          }
53284|
53285|          LocalHeader->Size = SizeInMemory;
53286|          LocalHeader->Version =
    | PSM_HEADER_CURRENT_VERSION;
53287|          LocalHeader->Signature =
    | PSM_HEADER_SIGNATURE | PSM_HEADER_MINOR_VERSION;
53288|
53289|          // We MUST calculate the header
    | checksum as the last step in constructing the header...
53290|          ULONG
    | *Checksum=(ULONG*)((((BYTE*)LocalHeader)+SizeInMemory-si
    | zeof(DWORD)));
53291|          *Checksum = CalculateChecksum
    | (SizeInMemory-sizeof(DWORD), LocalHeader);
53292|
53293|          // Rotated headers: Adjust location
    | based on the next place to put a header block...
53294|          ASSERT(NextHeaderBlockToSave >= 0);
53295|          ASSERT(NextHeaderBlockToSave <
    | NumSavedHeaderBlocks);
53296|          Location.QuadPart = BlockSizeOnDisk
    | * NextHeaderBlockToSave;
53297|
53298|          // Rotated headers: Save next
    | header block in another slot...
53299|          ASSERT (NumSavedHeaderBlocks != 0);
53300|          ULONG CurrentHeaderBlock =
    | NextHeaderBlockToSave;
53301|          NextHeaderBlockToSave = (1 +
    | NextHeaderBlockToSave) % NumSavedHeaderBlocks;
53302|          | Debug(DEBUG_DICT,("pd::SaveHeaderThread:
    | CurrentHeaderBlock=%x,
    | NextHeaderBlock=%x\n",CurrentHeaderBlock,NextHeaderBlock
    | ToSave));
53303|
53304|          if ( SnapShotAcquired ) {
53305|              ReleaseSnapShotForRead();

```

```

53306|             SnapShotAcquired = FALSE;
53307|         }
53308|
53309|             // we are breaking a rule here.
53310|             // which is that an io may not
53311|             | occur while we have the
53312|             // snapshot resource acquired from
53313|             | write (which during close)
53314|             // we do. This is a hack to allow
53315|             | it during close. This is
53316|             // somewhat safe as the file is not
53317|             | being extended so the
53318|             // filesystem doesnt need any
53319|             | volume resources to write to it.
53320|             // during open, when it may be
53321|             | extended we dont have the resource
53322|             // acquired for write, so we are
53323|             | safe.
53324|
53325|             Status= PsmWriteToFile(
53326|                 &DevExt->Cache.HeaderFile,
53327|                 &IoStatus,
53328|                 LocalHeader,
53329|                 SizeOnDisk,
53330|                 &Location,
53331|                 DevExt->DoDirectIO,
53332|                 | &DevExt->Cache.DirectAccessResource
53333|             #ifdef DEBUG
53334|                 ,FALSE
53335|             #endif
53336|             );
53337|
53338|             DumpHeader();
53339|             MemFreePool (LocalHeader);
53340|             LocalHeader = 0;
53341|         } else {
53342|             | Status=STATUS_INSUFFICIENT_RESOURCES;
53343|
53344|             | Debug(DEBUG_DICT,("pd::SaveHeaderThread: Error %08x
53345|             | allocating memory\n",Status));
53346|         }
53347|     } else {
53348|         | Debug(DEBUG_DICT,("pd::SaveHeaderThread:
53349|         | DevExt->Cache.Header is NULL.\n"));
53350|         ASSERT (TotalNumberOfSnapShots == 0);
53351|     }
53352| } __except(

```

```

    | ExceptionFilter(GetExceptionInformation()) ) {
53343|         Debug(DEBUG_DICT,("Exception %08x in
    | pd::SaveHeaderThread\n",GetExceptionCode()));
53344|     }
53345| } __finally {
53346|     if ( SnapShotAcquired ) {
53347|         ReleaseSnapShotForRead();
53348|         SnapShotAcquired = FALSE;
53349|     }
53350| }
53351|
53352| if ( Psh->ShouldSignalEvent ) {
53353|     // started as thread
53354|     pmSetEvent(&Psh->Event);
53355|     PsTerminateSystemThread( 0 );
53356| }
53357| }
53358|
53359| //-----
    | -----
53360|
53361| NTKERNELAPI
53362| BOOLEAN
53363| IolsSystemThread(
53364|     IN PETHREAD Thread
53365| );
53366|
53367| //-----
    | -----
53368|
53369| NTSTATUS PersistentDictionary::SaveHeader (
    | PDEVICE_OBJECT Volume )
53370| {
53371|     NTSTATUS Status = STATUS_SUCCESS;
53372|     __try {
53373|         if ( Updating == 0 ) {
53374|             PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
53375|             if(DevExt->Cache.Header) {
53376|                 #ifdef DEBUG
53377|                     PVOID CallerAddress=0,
    | CallersCallerAddress=0;
53378|                     RtlGetCallersAddress (&CallerAddress,
    | &CallersCallerAddress);
53379|                     Debug(DEBUG_DICT,("Entering
    | pd::SaveHeader: volume=%08x, caller=%08x,
    | grandpa=%08x\n",
53380|                         Volume,
53381|                         CallerAddress,
53382|                         CallersCallerAddress));

```



```

53383|         #endif
53384|
53385|         tSaveHeaderParameters Psh = {0};
53386|         Psh.Volume = Volume;
53387|         Psh.ShouldSignalEvent = FALSE;
53388|         KeInitializeEvent ( &Psh.Event,
| NotificationEvent, FALSE );
53389|
53390|         if(GlobalSystemProcessId ==
| PsGetCurrentProcess()) {
53391|             SaveHeaderThread(&Psh);
53392|         } else {
53393|             HANDLE ThreadHandle =
| INVALID_HANDLE_VALUE;
53394|             Psh.ShouldSignalEvent = TRUE;
53395|
53396|             pmStartThread(
53397|             | (PKSTART_ROUTINE)SaveHeaderThread,
53398|             | (PVOID)&Psh,
53399|             | &ThreadHandle );
53400|
53401|             | pmWaitForSingleObject(&Psh.Event,NULL);
53402|             ZwClose(ThreadHandle);
53403|         }
53404|
53405|         if ( NT_SUCCESS(Status) ) {
53406|             Debug(DEBUG_DICT,( "Successfully
| saved header\n"));
53407|         } else {
53408|             Debug(DEBUG_DICT,( "!!!
| SaveHeader() returning 0x%08lx\n", (unsigned
| long)Status ));
53409|         }
53410|         } else {
53411|             // nothing to do.
53412|             Debug(DEBUG_DICT,("pd::SaveHeader: No
| header to save - returning Status=%08x\n",Status));
53413|         }
53414|     }
53415| } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
53416|     Status = GetExceptionCode();
53417|     Debug(DEBUG_DICT,("!!! Exception %08x in
| pd::SaveHeader\n",Status));
53418| }
53419|
53420| return Status;
53421| }

```

```

53422|
53423|
53424| //-----
    | -----
53425|
53426|
53427| void PersistentDictionary::DumpHeader()
53428| {
53429| #if 0
53430|     Debug(DEBUG_DICT,( "Header:\n"));
53431|     Debug(DEBUG_DICT,( " Version      :
    | %08x\n",Header->Version));
53432|     Debug(DEBUG_DICT,( " Num SnapShots :
    | %08x\n",Header->NumberOfSnapShots));
53433|     Debug(DEBUG_DICT,( " Cache File   :
    | %S\n",Header->CacheFile));
53434|     Debug(DEBUG_DICT,( " Index File    :
    | %S\n",Header->IndexFile));
53435|     Debug(DEBUG_DICT,( " Granule Size  :
    | %08x\n",Header->GranuleSizeInBytes));
53436|     Debug(DEBUG_DICT,( " IdxLoadWheel  :
    | %08x\n",Header->IndexLoadWheel));
53437|     Debug(DEBUG_DICT,( " Highest Snap# :
    | %08x\n",Header->HighestSnapNumber));
53438|
53439|     ULONG Num=0,i;
53440|
53441|     Debug(DEBUG_DICT,( " BitMap Dump\n"));
53442| #define LINE_SIZE (32*8)
53443|     for ( i=0;i<MAX_NUMBER_OF_SNAPSHOTS;i+=LINE_SIZE )
    | {
53444|         Debug(DEBUG_DICT,( "
    | %08x-%08x-%08x-%08x-%08x-%08x-%08x-%08x\n",
53445|         | ((PRTL_BITMAP)&Header->SnapShotBitmap)->Buffer[i/LINE_SI
    | ZE],
53446|         | ((PRTL_BITMAP)&Header->SnapShotBitmap)->Buffer[i/LINE_SI
    | ZE+1],
53447|         | ((PRTL_BITMAP)&Header->SnapShotBitmap)->Buffer[i/LINE_SI
    | ZE+2],
53448|         | ((PRTL_BITMAP)&Header->SnapShotBitmap)->Buffer[i/LINE_SI
    | ZE+3],
53449|         | ((PRTL_BITMAP)&Header->SnapShotBitmap)->Buffer[i/LINE_SI
    | ZE+4],
53450|         | ((PRTL_BITMAP)&Header->SnapShotBitmap)->Buffer[i/LINE_SI

```

```

    | ZE+5],
53451|
    | ((PRTL_BITMAP)&Header->SnapShotBitmap)->Buffer[i/LINE_SI
    | ZE+6],
53452|
    | ((PRTL_BITMAP)&Header->SnapShotBitmap)->Buffer[i/LINE_SI
    | ZE+7]
53453|         ));
53454|     }
53455|
53456|     for ( i=0;i<MAX_NUMBER_OF_SNAPSHOTS;i++ ) {
53457|         PsmBitPositionValidate
    | ((PRTL_BITMAP)(Header->SnapShotBitmap), i);
53458|         ULONG InUse =
    | RtlCheckBit((PRTL_BITMAP)(Header->SnapShotBitmap),i);
53459|
53460|         if ( InUse ) {
53461|             Num++;
53462|         }
53463|
53464|         // dont print empty locations
53465|         if ( (Header->SnapShots[i].SequenceNumber!=0)
    | &&
53466|
    | (Header->SnapShots[i].SnapShotTime.QuadPart!=0) ) {
53467|             Debug(DEBUG_DICT,( " Snapshot number
    | %02x\n",i));
53468|             Debug(DEBUG_DICT,( " Marked in use :
    | %s\n",InUse ? "Yes" : "No"));
53469|             Debug(DEBUG_DICT,( " Sequence Number:
    | %08x\n",Header->SnapShots[i].SequenceNumber));
53470|             Debug(DEBUG_DICT,( " SnapShot Time :
    | %08x%08x\n",Header->SnapShots[i].SnapShotTime.HighPart,H
    | eader->SnapShots[i].SnapShotTime.LowPart));
53471|             Debug(DEBUG_DICT,( " Ext. Instance :
    | %08x\n",Header->SnapShots[i].ExternalInstance));
53472|             Debug(DEBUG_DICT,( " Caller Private :
    | %08x\n",Header->SnapShots[i].CallerPrivateUse));
53473|             Debug(DEBUG_DICT,( " Dll Private :
    | %08x\n",Header->SnapShots[i].DllPrivateUse));
53474|             Debug(DEBUG_DICT,( " Num to Keep :
    | %08x\n",Header->SnapShots[i].NumToKeep));
53475|             Debug(DEBUG_DICT,( " Priority :
    | %08x\n",Header->SnapShots[i].Priority));
53476|             Debug(DEBUG_DICT,( " SnapShot flags :
    | %08x\n",Header->SnapShots[i].SnapShotFlags));
53477|             Debug(DEBUG_DICT,( " SnapShot Name :
    | %S\n",Header->SnapShots[i].UserSnapShotName));
53478|         } else {
53479|             if ( InUse ) {

```

```

53480|         Debug(DEBUG_DICT,( "SnapShot %d is
| empty, but says it is in use\n",i));
53481| #ifdef DEBUG
53482|         DbgBreakPoint();
53483| #endif
53484|     }
53485| }
53486| }
53487| DumpRevertInfo ( Header->RevertInfo );
53488|
53489| if ( Num!=Header->NumberOfSnapShots ) {
53490|     Debug(DEBUG_DICT,( "Number of snapshots
| mismatch bitmap %08x != counter
| %08x\n",Num,Header->NumberOfSnapShots));
53491| #ifdef DEBUG
53492|     DbgBreakPoint();
53493| #endif
53494| }
53495| #endif
53496| }
53497|
53498| //-----
| -----
| ----
53499|
53500| NTSTATUS PersistentDictionary::FreeHeader (
| PDEVICE_OBJECT Volume )
53501| {
53502|     Profile("pd::FreeHeader");
53503|     NTSTATUS Status = STATUS_SUCCESS;
53504|
53505|     ASSERT (Volume != NULL);
53506|     if ( Volume ) {
53507|         PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
53508|
53509|         PVOID CallerAddress=0, CallersCallerAddress=0;
53510|         RtlGetCallersAddress (&CallerAddress,
| &CallersCallerAddress);
53511|         Debug(DEBUG_DICT,("Decrementing
| HeaderReferenceCount: Count=%08x, caller=%08x,
| grandpa=%08x\n",DevExt->Cache.HeaderReferenceCount,Calle
| rAddress,CallersCallerAddress));
53512|
53513|         if(DevExt->Cache.HeaderReferenceCount) {
53514|             if (
| --DevExt->Cache.HeaderReferenceCount==0 ) {
53515|                 ASSERT(DevExt->Cache.Header != NULL);
53516|                 if ( DevExt->Cache.Header != NULL ) {
53517|                     MemFreePool(DevExt->Cache.Header);

```

```

53518|         DevExt->Cache.Header = NULL;
53519|     }
53520| }
53521| } else {
53522|     ASSERT(DevExt->Cache.Header == NULL);
53523| }
53524| } else {
53525|     Status = STATUS_INVALID_PARAMETER;
53526| }
53527|
53528| return Status;
53529| }
53530|
53531| //-----
| -----
| ----
53532|
53533| NTSTATUS PersistentDictionary::LoadHeader (
| PDEVICE_OBJECT Volume )
53534| {
53535|     PFILTERED_EXTENSION DevExt = GetFilteredExtension
| (Volume);
53536|
53537|     // Extract from the header file the most recent
| header block that is valid (based on checksum).
53538|     pHeader MostRecentValidHeader=NULL,
| SmallMostRecentValidHeader=NULL;
53539|     NTSTATUS Status = ReadHeader (Volume,
| MostRecentValidHeader, SmallMostRecentValidHeader);
53540|     // Create all in-memory snapshot data structures
| based on what is in MostRecentValidHeader.
53541|     if ( MostRecentValidHeader!=NULL &&
| SmallMostRecentValidHeader!=NULL ) {
53542|         ASSERT(NT_SUCCESS(Status));
53543|         ULONG NumSnapShots =
| (MostRecentValidHeader->Size - sizeof(DWORD) -
| sizeof(tHeader)) / sizeof(tDiskInternalSnapShot);
53544|         Debug(DEBUG_DICT,("pd::LoadHeader: %d snapshots
| in list, header saved
| %016I64x\n",NumSnapShots,MostRecentValidHeader->DateTime
| Written.QuadPart));
53545|         for ( ULONG i=0; i<NumSnapShots; i++ ) {
53546|             pInternalSnapShot SnapShot =
| AllocSnapShot();
53547|             if ( SnapShot ) {
53548|                 SnapShot->SnapShotMaster=NULL;
53549|                 Debug(DEBUG_DICT,("pd::LoadHeader:
| Adding snapshot %d seq %d to snapshot
| list\n",i,MostRecentValidHeader->SnapShots[i].SequenceNu
| mber));

```

```

53550|
| RtlCopyMemory(&SnapShot->Permanent,&MostRecentValidHeader->Permanent,&MostRecentValidHeader->Permanent->Size);
| r->Snapshots[i],sizeof(tDiskInternalSnapshot));
53551|          // If header is before 2.1, the
| Status field will not be valid (it used to be
| DllPrivateUse pointer)
53552|          Debug(DEBUG_DICT,("pd::LoadHeader:
| Ver=%08x, Sig=%08x, SnapStatus=%08x\n",
53553|          MostRecentValidHeader->Version,
53554|          MostRecentValidHeader->Signature,
53555|          SnapShot->Permanent.Status));
53556|          if (
| MostRecentValidHeader->Version<=2 &&
| (MostRecentValidHeader->Signature &
| PSM_HEADER_MINOR_MASK) < 0x10000000 ) {
53557|          SnapShot->Permanent.Status =
| STATUS_SUCCESS; // patch invalid status loaded from
| old header
53558|          }
53559|          SnapShot->Permanent.SnapShotFlags =
| PSM_SS_HeaderToFlags
| (SnapShot->Permanent.SnapShotFlags);
53560|
| AddSnapShotToList(&DevExt->Cache.SnapShotHead,SnapShot);
53561|          SnapShot->ReferenceCount++;
53562|          DumpSnapShot(&SnapShot->Permanent);
53563|      } else {
53564|          Status = STATUS_INSUFFICIENT_RESOURCES;
53565|          Debug(DEBUG_DICT,( "pd::LoadHeader()
| error %08x allocating memory\n", Status ));
53566|      }
53567|  }
53568|  if ( DevExt->Cache.Header ) {
53569|      PVOID CallerAddress=0,
| CallersCallerAddress=0;
53570|      RtlGetCallersAddress (&CallerAddress,
| &CallersCallerAddress);
53571|      Debug(DEBUG_DICT,("Decrementing
| HeaderReferenceCount: Count=%08x, caller=%08x,
| grandpa=%08x\n",DevExt->Cache.HeaderReferenceCount,Calle
| rAddress,CallersCallerAddress));
53572|      ASSERT(DevExt->Cache.HeaderReferenceCount);
53573|      DevExt->Cache.HeaderReferenceCount--;
53574|      MemFreePool(DevExt->Cache.Header);
53575|  }
53576|  DevExt->Cache.Header =
| SmallMostRecentValidHeader;
53577|  SmallMostRecentValidHeader = NULL; // keep
| from freeing below
53578|  PVOID CallerAddress=0, CallersCallerAddress=0;

```

```

53579|     RtlGetCallersAddress (&CallerAddress,
| &CallersCallerAddress);
53580|     Debug(DEBUG_DICT,("Incrementing
| HeaderReferenceCount: Count=%08x, caller=%08x,
| grandpa=%08x\n",DevExt->Cache.HeaderReferenceCount,Calle
| rAddress,CallersCallerAddress));
53581|     DevExt->Cache.HeaderReferenceCount++;
53582| } else {
53583|     if((Status==STATUS_FILE_CORRUPT_ERROR ) || //
| corrupt header
53584|         (Status==STATUS_UNSUCCESSFUL)) //
| no header
53585|     {
53586|         Debug(DEBUG_DICT,( "pd::LoadHeader: Could
| not load header from disk, so setting to defaults\n"));
53587|
53588|         if ( SmallMostRecentValidHeader == NULL ) {
53589|             const ULONG HEADER_BLOCK_SIZE =
| ROUND_UP(sizeof(tHeader),DevExt->BytesPerSector);
53590|             SmallMostRecentValidHeader = (pHeader)
| MemAllocatePoolWithTag(PagedPool,HEADER_BLOCK_SIZE,PSM_D
| ICT_HEADER_TAG);
53591|         }
53592|         // new file, lets set up defaults.
53593|
| RtlZeroMemory(SmallMostRecentValidHeader,sizeof(tHeader)
| );
53594|         SmallMostRecentValidHeader->Version =
| (_BuildNumber_ << 16) | PSM_HEADER_CURRENT_VERSION;
53595|
| SmallMostRecentValidHeader->GranuleSizeInBytes =
| GRANULE_SIZE;
53596|         SmallMostRecentValidHeader->IndexLoadWheel
| = 0;
53597|         SmallMostRecentValidHeader->Size =
| sizeof(tHeader);
53598|
| SmallMostRecentValidHeader->HighestSnapNumber = 1;
53599|
53600|         ASSERT(DevExt->Cache.Header==NULL);
53601|         DevExt->Cache.Header =
| SmallMostRecentValidHeader;
53602|         SmallMostRecentValidHeader = NULL; // keep
| from freeing below
53603|         PVOID CallerAddress=0,
| CallersCallerAddress=0;
53604|         RtlGetCallersAddress (&CallerAddress,
| &CallersCallerAddress);
53605|         Debug(DEBUG_DICT,("Incrementing
| HeaderReferenceCount: Count=%08x, caller=%08x,

```

```

    | grandpa=%08x\n",DevExt->Cache.HeaderReferenceCount,Calle
    | rAddress,CallersCallerAddress));
53606|         DevExt->Cache.HeaderReferenceCount++;
53607|         Status = STATUS_SUCCESS;
53608|     } else {
53609|         Debug(DEBUG_DICT,( "pd::LoadHeader: Error
    | %08x reading header\n",Status));
53610| #ifdef DEBUG
53611|         switch(Status) {
53612|             case STATUS_DEVICE_OFF_LINE :
53613|             case STATUS_DEVICE_BUSY :
53614|             case STATUS_INSUFFICIENT_RESOURCES:
53615|                 break;
53616|             default:
53617|                 ASSERT(FALSE);
53618|         }
53619| #endif
53620|     }
53621| }
53622|
53623| if ( MostRecentValidHeader ) {
53624|     FREE_POINTER (MostRecentValidHeader);
53625| }
53626|
53627| if ( SmallMostRecentValidHeader ) {
53628|     FREE_POINTER (SmallMostRecentValidHeader);
53629| }
53630|
53631| Debug(DEBUG_DICT,("pd::LoadHeader returning
    | %08x\n",Status));
53632| return Status;
53633| }
53634|
53635| //-----
    | -----
    | ----
53636|
53637| NTSTATUS PersistentDictionary::ReadHeader (
53638|     PDEVICE_OBJECT    Volume,
    | // IN: which volume to read header for
53639|     pHeader            &MostRecentValidHeader,
    | // OUT: if not NULL, latest valid header read
53640|     pHeader            &SmallMostRecentValidHeader )
    | // OUT: if not NULL, latest header prefix read
53641| {
53642|     NTSTATUS BlockStatus=STATUS_UNSUCCESSFUL;
53643|
53644|     Profile("pd::LoadHeader");
53645|     Debug(DEBUG_DICT,("pd::LoadHeader:
    | Volume=%08x\n",Volume));

```





```

53680|         | FREE_POINTER(SmallMostRecentValidHeader);
53681|         }
53682|         if ( MostRecentValidHeader ) {
53683|         | FREE_POINTER(MostRecentValidHeader);
53684|         }
53685|         SmallMostRecentValidHeader =
53686|         | SmallLocalHeader;
53687|         MostRecentValidHeader =
53688|         | LocalHeader;
53689|         SmallLocalHeader = NULL; // keep
53690|         | from freeing below
53691|         LocalHeader = NULL;
53692|     }
53693| } else {
53694|     ASSERT(SmallLocalHeader == NULL);
53695|     ASSERT(LocalHeader == NULL);
53696| }
53697| if ( SmallLocalHeader ) {
53698|     FREE_POINTER (SmallLocalHeader);
53699| }
53700|
53701| if ( LocalHeader ) {
53702|     FREE_POINTER (LocalHeader);
53703| }
53704| }
53705|
53706| NTSTATUS Status = (MostRecentValidHeader &&
53707| | SmallMostRecentValidHeader) ? STATUS_SUCCESS :
53708| | BlockStatus;
53709| Debug(DEBUG_DICT,("pd::ReadHeader returning
53710| | %08x\n",Status));
53711| return Status;
53712| }
53713| //-----
53714| | -----
53715| | ----
53716|
53717| NTSTATUS PersistentDictionary::ReadHeaderBlock (
53718| PDEVICE_OBJECT Volume,
53719| ULONG HeaderBlockIndex,
53720| pHeader &SmallLocalHeader,
53721| pHeader &LocalHeader )
53722| {
53723| SmallLocalHeader = NULL;

```

```

53720|   LocalHeader = NULL;
53721|
53722|   PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
53723|   const ULONG HEADER_BLOCK_SIZE =
    | ROUND_UP(sizeof(tHeader),DevExt->BytesPerSector);
53724|   NTSTATUS Status = STATUS_SUCCESS;
53725|   IO_STATUS_BLOCK IoStatus = {0};
53726|   LARGE_INTEGER Location = {0};
53727|
53728|   Profile("pd::ReadHeaderBlock");
53729|   Debug(DEBUG_DICT,("pd::ReadHeaderBlock:
    | Volume=%08x,
    | HeaderBlockIndex=%08x\n",Volume,HeaderBlockIndex));
53730|
53731|   SmallLocalHeader = (pHeader)
    | MemAllocatePoolWithTag(PagedPool,HEADER_BLOCK_SIZE,PSM_D
    | ICT_HEADER_TAG);
53732|   if ( SmallLocalHeader ) {
53733|       ULONG BlockSizeInMemory =
    | sizeof(tHeader)+(MAX_SNAPSHOTS_PER_HEADER_FILE*sizeof(tD
    | iskInternalSnapShot))+sizeof(DWORD);
53734|       ULONG BlockSizeOnDisk =
    | ROUND_UP(BlockSizeInMemory,DevExt->BytesPerSector);
53735|       ASSERT(BlockSizeOnDisk>=BlockSizeInMemory);
53736|       ASSERT(BlockSizeOnDisk % DevExt->BytesPerSector
    | == 0);
53737|       Location.QuadPart = HeaderBlockIndex *
    | BlockSizeOnDisk;
53738|       LARGE_INTEGER SaveLocation = Location;
53739|
53740|       Status = PsmReadFromFile(
53741|           &DevExt->Cache.HeaderFile,
53742|           &IoStatus,
53743|           SmallLocalHeader,
53744|           HEADER_BLOCK_SIZE,
53745|           &Location,
53746|           DevExt->DoDirectIO,
53747|           &DevExt->Cache.DirectAccessResource );
53748|
53749|       if ( NT_SUCCESS(Status) ) {
53750|           if ( SmallLocalHeader->Version ==
    | PSM_HEADER_CURRENT_VERSION ) {
53751|               if ( (SmallLocalHeader->Signature &
    | PSM_HEADER_SIGNATURE_MASK) == PSM_HEADER_SIGNATURE ) {
53752|                   ULONG Size =
    | SmallLocalHeader->Size;
53753|                   ULONG SizeOnDisk =
    | ROUND_UP(Size,DevExt->BytesPerSector);
53754|                   LocalHeader = (pHeader)

```

```

    | MemAllocatePoolWithTag(PagedPool,SizeOnDisk,PSM_DICT_HEA
    | DER_TAG);
53755|         if ( LocalHeader ) {
53756|             ASSERT (Location.QuadPart ==
    | SaveLocation.QuadPart);
53757|             Location.QuadPart =
    | SaveLocation.QuadPart;
53758|
53759|             Status = PsmReadFromFile(
53760|                 &DevExt->Cache.HeaderFile,
53761|                 &IoStatus,
53762|                 LocalHeader,
53763|                 SizeOnDisk,
53764|                 &Location,
53765|                 DevExt->DoDirectIO,
53766|
    | &DevExt->Cache.DirectAccessResource );
53767|
53768|             if ( NT_SUCCESS(Status) ) {
53769|                 ULONG
    | Checksum=*(ULONG*)((((BYTE*)LocalHeader)+Size-sizeof(DWO
    | RD)));
53770|                 // we already did version
    | checks, so dont do it again
53771|                 BOOLEAN isValid =
    | ValidateChecksum (Size-sizeof(DWORD), LocalHeader,
    | Checksum);
53772|                 if ( isValid ) {
53773|                     Status =
    | STATUS_SUCCESS;
53774|
    | Debug(DEBUG_DICT,("pd::ReadHeaderBlock - success
    | loading header index=%d,
    | time=%016l64x\n",HeaderBlockIndex,LocalHeader->DateTimew
    | ritten.QuadPart));
53775|                 } else {
53776|                     Status =
    | STATUS_FILE_CORRUPT_ERROR;
53777|                     Debug(DEBUG_DICT,(
    | "pd::ReadHeaderBlock() error %08x checksum doesnt
    | match\n", Status ));
53778|                 }
53779|             } else {
53780|                 Debug(DEBUG_DICT,(
    | "pd::ReadHeaderBlock() error %08x reading header\n",
    | Status ));
53781|             }
53782|         } else {
53783|             Status =
    | STATUS_INSUFFICIENT_RESOURCES;

```

```

53784|         Debug(DEBUG_DICT,(
    | "pd::ReadHeaderBlock() error %08x allocating memory\n",
    | Status ));
53785|     }
53786| } else {
53787|     Status = STATUS_FILE_CORRUPT_ERROR;
53788|     Debug(DEBUG_DICT,(
    | "pd::ReadHeaderBlock() signature %08x isnt ours\n",
    | SmallLocalHeader->Signature));
53789| }
53790| } else {
53791|     Status = STATUS_FILE_CORRUPT_ERROR;
53792|     Debug(DEBUG_DICT,(
    | "pd::ReadHeaderBlock() version %08x isnt ours\n",
    | SmallLocalHeader->Version));
53793| }
53794| } else {
53795|     Debug(DEBUG_DICT,( "pd::ReadHeaderBlock:
    | error %08x reading header\n", Status ));
53796| }
53797| } else {
53798|     Status = STATUS_INSUFFICIENT_RESOURCES;
53799|     Debug(DEBUG_DICT,( "pd::LoadHeader() error %08x
    | allocating memory\n", Status ));
53800| }
53801|
53802| // This function must return with one of the
    | following two postconditions:
53803| //
53804| // 1. Status is successful and both 'LocalHeader'
    | and 'SmallLocalHeader' point to
53805| // valid header information. The caller is
    | responsible for using/freeing the
53806| // two pointers when appropriate.
53807| //
53808| // 2. Status is unsuccessful and both 'LocalHeader'
    | and 'SmallLocalHeader' are NULL.
53809| //
53810| // In other words, the caller must be able to rely
    | upon the Status returned to tell
53811| // whether it has responsibility for the pointers.
53812|
53813| if ( NT_SUCCESS(Status) ) {
53814|     ASSERT (LocalHeader != NULL);
53815|     ASSERT (SmallLocalHeader != NULL);
53816| } else {
53817|     if ( LocalHeader ) {
53818|         MemFreePool(LocalHeader);
53819|         LocalHeader = NULL;
53820|     }

```

```

53821|
53822|     if ( SmallLocalHeader ) {
53823|         MemFreePool(SmallLocalHeader);
53824|         SmallLocalHeader = NULL;
53825|     }
53826| }
53827|
53828|     Debug(DEBUG_DICT,("pd::ReadHeaderBlock returning
| Status=%08x, LocalHeader=%08x,
| SmallLocalHeader=%08x\n",Status,LocalHeader,SmallLocalHe
| ader));
53829|     return Status;
53830| }
53831|
53832| //-----
| -----
53833|
53834| ErrorCode PersistentDictionary::SetVolumeInternal (
53835|     PDEVICE_OBJECT    AVolume,
53836|     pInternalSnapShot  MySnapShot,
53837|     ULONG              SnapShotSequence )
53838| {
53839|     pDictionary p=NULL;
53840|     NTSTATUS Status=STATUS_SUCCESS;
53841|
53842|     __try {
53843|         GetDictionaryForVolume(AVolume,p);
53844|         if ( p ) {
53845|             // share the same tree and resource with
| the other snapshots.
53846|             Shared =
| ((pPersistentDictionary)p)->Shared;
53847|             Shared->Count++;
53848|
53849|             Debug(DEBUG_DICT,("pd::SetVolumeInternal
| (dict=%08x): Incremented Shared->Count to %08x\n",
53850|                 this,
53851|                 Shared->Count));
53852|
53853|             // Hide the map while we update it. This
| will simulate as if no freespace check algorithm
| exists.
53854|             // The worst that can happen elsewhere is
53855|             // .1. we might not credit a granule we
| snap .. BUT.. The tree will protect us against taking a
| second snap for the same snapshot.
53856|             // .2. we might snap some free space we
| didn't need to!!
53857|             Shared->MapInTransform = Shared->Map;
53858|             Shared->Map = NULL;

```

```

53859|
53860|     } else {
53861|         Shared =
53862|         | (pShared)MemAllocatePoolWithTag(NonPagedPool,sizeof(tSha
53863|         | red),PSM_DICT_SHARED_TAG);
53864|         if ( Shared ) {
53865|             RtlZeroMemory ( Shared, sizeof(tShared)
53866|             | );
53867|             ExInitializeResourceLite(&Shared->TreeResource);
53868|             pmRegisterObject(&Shared->TreeResource,"Shared->TreeReso
53869|             | urce",pmRwLock);
53870|             rbtree_Init(&Shared->Tree);
53871|             rbtree_Init(&Shared->VirtualWritesTree);
53872|             Shared->MapInTransform = Shared->Map =
53873|             | NULL;
53874|             Shared->Count = 1;
53875|             Shared->RecycleNeeded = 0;
53876|             Shared->LastDirtyKey = 0;
53877|             Shared->HighestKeyKnownClean = 0;
53878|             Shared->HighestSequence = 0; // will
53879|             | be set to correct value below
53880|             | Debug(DEBUG_DICT,("pd::SetVolumeInternal (dict=%08x):
53881|             | Initialized Shared->Count to
53882|             | %08x\n",this,Shared->Count));
53883|         } else {
53884|             Status = STATUS_INSUFFICIENT_RESOURCES;
53885|             Debug(DEBUG_DICT,("!!!
53886|             | pd::SetVolumeInternal (dict=%08x): Out of memory for
53887|             | Shared\n",this));
53888|         }
53889|     }
53890|     SnapShot = MySnapShot;
53891|     if ( SnapShot ) {
53892|         if ( SnapShotSequence >
53893|         | Shared->HighestSequence ) {
53894|             Debug(DEBUG_DICT,("pd::SetVolumeInternal (dict=%08x):
53895|             | Changing Shared->HighestSequence from %08x to
53896|             | %08x\n",this,Shared->HighestSequence,SnapShotSequence));
53897|             Shared->HighestSequence =
53898|             | SnapShotSequence;
53899|         }

```

```

53890|     }
53891|
53892|     Volume = AVolume;
53893|     if ( Volume ) {
53894|         DevExt = GetFilteredExtension(Volume);
53895|     }
53896|
53897| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
53898|     Status = GetExceptionCode();
53899|     Debug(DEBUG_DICT,("Exception %08x in
    | pd::SetVolume\n",Status));
53900| }
53901|
53902| return Status;
53903| }
53904|
53905| //-----
    | -----
53906|
53907| ErrorCode PersistentDictionary::SetVolume (
53908|     PDEVICE_OBJECT    AVolume,
53909|     pkSnapShotMaster  MasterSnapShot,
53910|     ULONG              SnapShotSequence )
53911| {
53912|     pDictionary p=NULL;
53913|     NTSTATUS Status=STATUS_SUCCESS;
53914|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(AVolume);
53915|
53916|     __try {
53917|         //Use a time change as indicator a new snapshot
    | has occurred - (otherwise we'll use the same index
53918|         //thus keeping multi-volume snapshots linked.
53919|         //Relies on assumption that time is granular
    | enough to distinguish separate snapshots
53920|         LARGE_INTEGER Time =
    | MasterSnapShot->SnapShotTime;
53921|
53922|         if ( DevExt->Cache.MostRecentSnapshotTime !=
    | (ULONGLONG)Time.QuadPart ) {
53923|             DevExt->Cache.MostRecentSnapshot =
    | GetNextFreeSnapShot (AVolume, SnapShotSequence);
53924|             DevExt->Cache.MostRecentSnapshotTime =
    | Time.QuadPart;
53925|         } else {
53926|             if ( DevExt->Cache.MostRecentSnapshot ) {
53927|                 | DevExt->Cache.MostRecentSnapshot->ReferenceCount++;
53928|             }

```



```

53929|     }
53930|     SnapShot = DevExt->Cache.MostRecentSnapshot;
53931|     if ( SnapShot ) {
53932|         Status =
53933|         | SetVolumeInternal(AVolume,SnapShot,SnapShotSequence);
53934|         if ( NT_SUCCESS(Status) ) {
53935|             SnapShot->Permanent.SnapShotTime =
53936|             | Time;
53937|             SnapShot->SnapShotMaster =
53938|             | MasterSnapShot;
53939|             // Dont save header here as we can not
53940|             | do io or a deadlock will
53941|             | occur. It is easier to not do it
53942|             | here, and in the SetInstance
53943|             // save the header. If we really need
53944|             | to save here then
53945|             // we need to change the high level
53946|             | code so it doesnt call us with
53947|             // io disabled
53948|             //      Status = SaveHeader();
53949|             //      VerifyHeaderIntegrity ( Header,
53950|             | "global header after SetVolume" );
53951|             } else {
53952|                 Debug(DEBUG_DICT,("pd::SetVolume:
53953|                 | SetVolumeInternal Failed %08x",Status));
53954|                 ASSERT(SnapShot->ReferenceCount);
53955|                 if ( --SnapShot->ReferenceCount==0 ) {
53956|                     RemoveSnapShotFromList(SnapShot);
53957|                     FreeSnapShot(&SnapShot);
53958|                 }
53959|                 Header->NumberOfSnapShots--;
53960|                 //
53961|                 | RtlClearBits((PRTL_BITMAP)(Header->SnapShotBitmap),SnapS
53962|                 | hotIndex,0);
53963|             }
53964|             } else {
53965|                 Status = STATUS_INSUFFICIENT_RESOURCES;
53966|                 Debug(DEBUG_DICT,("pd::SetVolume: No free
53967|                 | snapshot bits"));
53968|             }
53969|             } __except(
53970|             | ExceptionFilter(GetExceptionInformation()) ) {
53971|                 Status = GetExceptionCode();
53972|                 Debug(DEBUG_DICT,("Exception %08x in
53973|                 | pd::SetVolume\n",Status));
53974|             }
53975|             }
53976|             return Status;
53977|         }

```

```

53965|
53966| //-----
| -----
53967|
53968| ErrorCode
| PersistentDictionary::UpdateCacheFileSizes(PDEVICE_OBJEC
| T Volume)
53969| {
53970|     NTSTATUS Status = STATUS_UNSUCCESSFUL;
53971|     #if 0
53972|         ULONG NewInitialSize=0;
53973|         ULONG NewMaxSize=0;
53974|         ULONG NewMaxSizeG=0;
53975|         ULONG NewInitialSizeG=0;
53976|         FILE_END_OF_FILE_INFORMATION EOFInfo={0};
53977|         IO_STATUS_BLOCK IoStatus={0};
53978|         WCHAR Buffer[255]={0};
53979|         UNICODE_STRING Reg={0};
53980|         PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
53981|
53982|         Debug(DEBUG_DICT,("pd::UpdateCacheFileSizes;
| Volume=%08x, DevExt=%08x, Header=%08x\n",
53983|             Volume,
53984|             DevExt,
53985|             DevExt->Cache.Header));
53986|
53987|         if ( DevExt->Cache.Header ) {
53988|             ASSERT(gRegistryPath.Length<sizeof(Buffer));
53989|
53990|             | RtlCopyMemory(Buffer,gRegistryPath.Buffer,gRegistryPath.
| Length);
53991|
53992|             // NULL terminate, since counted unicode
| strings are not necessarily null
53993|             // terminated
53994|             Buffer[gRegistryPath.Length / 2] = 0;
53995|
53996|             wcscat(Buffer,L"\");
53997|             wcscat(Buffer,DevExt->VolumeGuid);
53998|
53999|             RtlInitUnicodeString(&Reg,Buffer);
54000|
54001|             | Reg_GetULONGKey(&Reg,L"InitialSize",10,&NewInitialSize);
54002|             | Reg_GetULONGKey(&Reg,L"MaxSize",100,&NewMaxSize);
54003|
54004|             NewMaxSizeG =

```

```

    | NewMaxSize*((1024*1024)/GRANULE_SIZE);
54005|     NewInitialSizeG =
    | NewInitialSize*((1024*1024)/GRANULE_SIZE);
54006|
54007|     ASSERT ( NewMaxSizeG % sizeof(ULONG) == 0 );
54008|     ASSERT ( NewInitialSizeG % sizeof(ULONG) == 0
    | );
54009|
54010|     if (
    | NewMaxSizeG>DevExt->Cache.PSManBitMapMaxSize ) {
54011|
54012|         // allocate outside of bitmap mutex, as io
    | may be generated
54013|         // when we allocate memory
54014|         PRTL_BITMAP NewBuffer = (PRTL_BITMAP)
    | MemAllocatePoolWithTag( PagedPool,
    | sizeof(RTL_BITMAP)+(NewMaxSizeG) / 8, BITMAPTAG);
54015|
54016|         // allocate new bitmap
54017|         pmAcquireMutex (
    | &DevExt->Cache.PSManBitMapMutex, NULL );
54018|         __try {
54019|             if ( NewBuffer!= NULL ) {
54020|
    | RtlInitializeBitMap(NewBuffer,(PULONG)((char*)(NewBuffer
    | )+sizeof(RTL_BITMAP)),NewMaxSizeG);
54021|             RtlClearAllBits(NewBuffer);
54022|
    | RtlMoveMemory(NewBuffer->Buffer,DevExt->Cache.PSManBitMa
    | pBuffer->Buffer,DevExt->Cache.PSManBitMapMaxSize / 8);
54023|             PVOID Temp =
    | DevExt->Cache.PSManBitMapBuffer;
54024|             DevExt->Cache.PSManBitMapBuffer =
    | NewBuffer;
54025|             MemFreePool(Temp);
54026|             DevExt->Cache.PSManBitMapMaxSize =
    | NewMaxSizeG;
54027|         } else {
54028|             // cant set new max, lets try and
    | move the initial as high
54029|             // up as we can
54030|             if (
    | NewInitialSizeG>DevExt->Cache.PSManBitMapMaxSize ) {
54031|                 NewInitialSizeG =
    | DevExt->Cache.PSManBitMapMaxSize;
54032|             }
54033|         }
54034|     } __finally {
54035|         pmReleaseMutex
    | (&DevExt->Cache.PSManBitMapMutex);

```

```

54036|         }
54037|     } else {
54038|         // new max size is smaller than last time..
54039|         if ( NewMaxSizeG <
            | DevExt->Cache.PSManBitMapSize ) {
54040|             DevExt->Cache.PSManBitMapMaxSize =
            | DevExt->Cache.PSManBitMapSize;
54041|         } else {
54042|             DevExt->Cache.PSManBitMapMaxSize =
            | NewMaxSizeG;
54043|         }
54044|     }
54045|     if (
            | NewInitialSizeG>DevExt->Cache.PSManBitMapSize ) {
54046|         DevExt->DoDirectIO = TRUE;
54047|         EOFInfo.EndOfFile.QuadPart = (unsigned
            | _int64)NewInitialSizeG*GRANULE_SIZE;
54048|
54049|         Status = ZwSetInformationFile(
54050|             | DevExt->Cache.CacheFile.FileHandle,          //
            | IN_HANDLE FileHandle,
54051|             &IoStatus,
            | // OUT PIO_STATUS_BLOCK IoStatusBlock,
54052|             &EOFInfo,
            | // IN PVOID FileInformation,
54053|
            | sizeof(EOFInfo),          // IN ULONG Length,
54054|
            | FileEndOfFileInformation // IN FILE_INFORMATION_CLASS
            | FileInformationClass
54055|             );
54056|
54057|         EOFInfo.EndOfFile.QuadPart = (unsigned
            | _int64)NewInitialSizeG*SectorSize;
54058|
54059|         Status = ZwSetInformationFile(
54060|             | DevExt->Cache.IndexFile.FileHandle,          //
            | IN_HANDLE FileHandle,
54061|             &IoStatus,
            | // OUT PIO_STATUS_BLOCK IoStatusBlock,
54062|             &EOFInfo,
            | // IN PVOID FileInformation,
54063|
            | sizeof(EOFInfo),          // IN ULONG Length,
54064|
            | FileEndOfFileInformation // IN FILE_INFORMATION_CLASS
            | FileInformationClass
54065|             );

```

```

54066|         DevExt->Cache.PSManBitMapSize =
| NewInitialSizeG;
54067|         DevExt->DoDirectIO = FALSE;
54068|     } else {
54069|         // hmm, we cant shrink the current
54070|         // window because data could be spread out
| across
54071|         // the cache file, so leave it at its
| current
54072|         // size
54073|     }
54074| }
54075| #endif
54076| return Status;
54077| }
54078|
54079| //-----
| -----
| -----
54080|
54081| ErrorCode PersistentDictionary::BeginUpdate()
54082| {
54083|     ASSERT (Updating >= 0);
54084|     InterlockedIncrement(&Updating);
54085|     return STATUS_SUCCESS;
54086| }
54087|
54088| //-----
| -----
| -----
54089|
54090| ErrorCode PersistentDictionary::EndUpdate()
54091| {
54092|     NTSTATUS Status = STATUS_SUCCESS;
54093|
54094|     signed long value =
| InterlockedDecrement(&Updating);
54095|     ASSERT(value>=0);
54096|     if(value==0) {
54097|         PDEVICE_OBJECT DevObj =
| PSMANDriverObject->DeviceObject;
54098|         while ( DevObj != NULL ) {
54099|             if (
| PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
54100|                 NTSTATUS SaveStatus =
| SaveHeader(DevObj);
54101|                 if ( NT_SUCCESS(SaveStatus) ) {
54102|                     UpdateCacheFileSizes(DevObj);
54103|                     //VerifyHeaderIntegrity ( Header,
| "Global header after EndUpdate" );

```

```

54104|         } else {
54105|             Debug(DEBUG_DICT,("pd::EndUpdate:
| SaveHeader on DevObj=%08x returned
| SaveStatus=%08x\n",DevObj,SaveStatus));
54106|             ASSERT(NT_SUCCESS(SaveStatus)); //
| temporary experiment
54107|             if ( NT_SUCCESS(Status) ) {
54108|                 Status = SaveStatus;
54109|             }
54110|         }
54111|     }
54112|
54113|     DevObj=DevObj->NextDevice;
54114| }
54115| }
54116|
54117| return Status;
54118| }
54119|
54120| //-----
| -----
| -----
54121|
54122| ErrorCode PersistentDictionary::SetSnapShotInfo(
| pkSnapShotMaster Master )
54123| {
54124|     pDictionary p=NULL;
54125|     NTSTATUS Status=STATUS_SUCCESS;
54126|
54127|     __try {
54128|         SnapShot->DllPrivateUse =
| Master->DllPrivateUse;
54129|
54130|         SnapShot->Permanent.GroupNumber      =
| Master->GroupNumber;
54131|         SnapShot->Permanent.ExternalInstance =
| Master->Instance;
54132|         SnapShot->Permanent.NumToKeep        =
| Master->NumToKeep;
54133|         SnapShot->Permanent.Priority          =
| Master->Priority;
54134|         SnapShot->Permanent.SnapShotFlags    =
| Master->SnapShotFlags;
54135|
54136|         wcsncpy(SnapShot->Permanent.UserSnapShotName,Master->User
| SnapShotName);
54137|
54138|         ASSERT (Updating >= 0);
54139|         if ( Updating == 0 ) {

```

```

54140|         Status = SaveHeader(Volume);
54141|
54142|         if ( NT_SUCCESS(Status) ) {
54143|             Status = UpdateCacheFileSizes(Volume);
54144|             //VerifyHeaderIntegrity ( Header,
| "global header after SetSnapShotInfo" );
54145|         }
54146|     }
54147|
54148| } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
54149|     Status = GetExceptionCode();
54150|     Debug(DEBUG_DICT,("Exception %08x in
| pd::SetInstance\n",Status));
54151| }
54152|
54153| return Status;
54154| }
54155|
54156| //-----
| -----
54157|
54158| ErrorCode PersistentDictionary::GetRegistrySettings (
| IN PUNICODE_STRING RegistryPath )
54159| {
54160|     WCHAR Buffer[255]={0};
54161|     UNICODE_STRING Reg={0};
54162|     NTSTATUS Status=0;
54163|     UNICODE_STRING Uni;
54164|
54165|     PAGED_CODE();
54166|
54167|     // get system start options
54168|
| wcsncpy(Buffer,L"\\Registry\\Machine\\System\\CurrentCont
| rolSet\\Control");
54169|     RtlInitUnicodeString(&Reg,Buffer);
54170|
54171|     Reg_GetStringKey (&Reg, L"SystemStartOptions",
| L"FASTDETECT", &Uni);
54172|     Debug(DEBUG_DICT,("pd::GetRegistrySettings:
| SystemStartOptions = '%S'\n",Uni.Buffer));
54173|     NoPsm = FALSE;
54174|     ResetPsm = FALSE;
54175|     if ( wcsstr(Uni.Buffer,L"NOPSM")!=NULL ) {
54176|         NoPsm = TRUE;
54177|     }
54178|     if ( wcsstr(Uni.Buffer,L"RESETPSM")!=NULL ) {
54179|         ResetPsm = TRUE;
54180|     }

```

```

54181|   Reg_FreeString(&Uni);
54182|   Debug(DEBUG_DICT,("pd::GetRegistrySettings:
| ResetPsm=%x, NoPsm=%x\n",ResetPsm,NoPsm));
54183|
54184|   ASSERT(RegistryPath->Length<sizeof(Buffer));
54185|
54186|
| RtlCopyMemory(Buffer,RegistryPath->Buffer,RegistryPath->
| Length);
54187|
54188|   // NULL terminate, since counted unicode strings
| are not necessarily null
54189|   // terminated
54190|   Buffer[RegistryPath->Length / 2] = 0;
54191|
54192|   wcscat(Buffer,L"\\persistent");
54193|
54194|   RtlInitUnicodeString(&Reg,Buffer);
54195|
54196|   Reg_GetStringKey (
| &Reg,L"SnapShotLocation",L"snapshots",&Uni);
54197|   wcscpy(gSnapShotDirName,Uni.Buffer);
54198|   Reg_FreeString(&Uni);
54199|
54200|   Reg_GetULONGKey(&Reg,L"QWait",5,&QPeriod);
54201|   Reg_GetULONGKey(&Reg,L"QTimeout",900,&QTimeout);
54202|   Reg_GetULONGKey(&Reg,L"ShareFiles",0,&ShareFiles);
54203|
| Reg_GetULONGKey(&Reg,L"MaxSnapShots",MAX_USER_SNAPSHOTS,
| &MaxNumUserSnapShots);
54204|
54205|   Debug(DEBUG_DICT,("pd::GetRegistrySettings:
| QPeriod=%08x, QTimeout=%08x, ShareFiles=%08x,
| MaxNumUserSnapShots=%08x\n",
54206|       QPeriod,
54207|       QTimeout,
54208|       ShareFiles,
54209|       MaxNumUserSnapShots));
54210|
54211|   if ( MaxNumUserSnapShots > MAX_USER_SNAPSHOTS ) {
54212|       MaxNumUserSnapShots = MAX_USER_SNAPSHOTS;
54213|   }
54214|
54215|   return(Status);
54216| }
54217|
54218| //-----
| -----
54219|
54220| ErrorCode

```



```

54221| PersistentDictionary::OpenAFile(
54222|     WCHAR          *File,
54223|     HANDLE          &FileHandle,
54224|     PFILE_OBJECT    &FileObject,
54225|     HANDLE          &WaitHandle,
54226|     PVOID           &WaitObject )
54227| {
54228|     UNICODE_STRING  FullFileName={0};
54229|     OBJECT_ATTRIBUTES ObjectAttributes={0};
54230|     NTSTATUS         Status=STATUS_UNSUCCESSFUL;
54231|     IO_STATUS_BLOCK  IoStatus={0};
54232|     ULONG            ShareAccess = 0;
54233|
54234|     PAGED_CODE();
54235|
54236|     Profile("pd::OpenAFile");
54237|     Debug(DEBUG_DICT,("pd::OpenAFile(%S)\n",File));
54238|
54239|     RtlInitUnicodeString( &FullFileName, File);
54240|
54241|     InitializeObjectAttributes ( &ObjectAttributes,
54242|                                   &FullFileName,
54243|                                   OBJ_CASE_INSENSITIVE,
54244|                                   NULL,
54245|                                   NULL );
54246|
54247|     if ( ShareFiles ) {
54248|         ShareAccess = FILE_SHARE_READ |
54249|         | FILE_SHARE_WRITE;
54250|     } else {
54251|         ShareAccess = 0;
54252|     }
54253|     Status = ZwCreateFile( &FileHandle,
54254|                           FILE_GENERIC_READ |
54255|         | FILE_GENERIC_WRITE, // desired access
54256|                           &ObjectAttributes, // object
54257|         | attributes
54258|                           &IoStatus,
54259|                           NULL,           //
54260|         | alloc size
54261|                           FILE_ATTRIBUTE_HIDDEN,
54262|         | // file attributes
54263|                           ShareAccess,
54264|         | // share access
54265|                           FILE_OPEN, //
54266|         | FILE_OVERWRITE_IF,         // create
54267|         | disposition
54268|                           FILE_SYNCHRONOUS_IO_NONALERT
54269|         | |

```

```

54262|          FILE_NO_COMPRESSION |
54263|          FILE_WRITE_THROUGH |
54264|          | FILE_NO_INTERMEDIATE_BUFFERING,
54265|          NULL, // eabuffer
54266|          0 ); // ealength
54267|
54268|
54269|  if ( NT_SUCCESS(Status) ) {
54270|      // Get a Object handle so we can wait on
      | requests...
54271|      Status = ObReferenceObjectByHandle(
54272|          FileHandle,
      | // IN HANDLE Handle,
54273|
      | FILE_GENERIC_READ | FILE_GENERIC_WRITE,
54274|          NULL,
      | // IN POBJECT_TYPE ObjectType,          // optional
54275|
      | (KPROCESSOR_MODE)KernelMode,          // IN
      | KPROCESSOR_MODE AccessMode,
54276|          (PVOID
      | *)&FileObject, // OUT PVOID *Object,
54277|          NULL
      | // OUT POBJECT_HANDLE_INFORMATION HandleInformation
      | // optional
54278|          );
54279|
54280|      if ( NT_SUCCESS(Status) ) {
54281|          Status =
      | SbGetAsyncEvent(WaitHandle,WaitObject);
54282|          if ( NT_SUCCESS(Status) ) {
54283|              ASSERT(IsValidHandle(FileHandle));
54284|              ASSERT(IsValidHandle(WaitHandle));
54285|              ASSERT(FileObject != NULL);
54286|              ASSERT(WaitObject != NULL);
54287|              return STATUS_SUCCESS;
54288|          } else {
54289|              Debug(DEBUG_DICT,("pd::OpenAFile:
      | SbGetAsyncEvent(WaitHandle,WaitObject) returned
      | %08x\n",Status));
54290|              if ( FileObject != NULL ) {
54291|                  ObDereferenceObject (FileObject);
54292|                  FileObject = NULL;
54293|              }
54294|          }
54295|      } else {
54296|          Debug(DEBUG_DICT,("pd::OpenAFile:
      | ObReferenceObjectByHandle() returned %08x\n",Status));
54297|      }

```

```

54298|
54299|     ZwClose(FileHandle);
54300|     FileHandle = INVALID_HANDLE_VALUE;
54301| } else {
54302|     Debug(DEBUG_DICT,("pd::OpenAFile:
    | ZwCreateFile() returned %08x\n",Status));
54303| }
54304|
54305| // Getting here means an error occurred.
54306| ASSERT(INT_SUCCESS(Status));
54307| ASSERT(FileHandle == INVALID_HANDLE_VALUE);
54308| ASSERT(WaitHandle == INVALID_HANDLE_VALUE);
54309| ASSERT(FileObject == NULL);
54310| ASSERT(WaitObject == NULL);
54311|
54312| Debug(DEBUG_DICT,("pd::OpenAFile: Error! Unable to
    | open file '%S'\n",File));
54313| Debug(DEBUG_DICT,(" Status=%08x,
    | IoStatus.Status=%08x\n",Status,IoStatus.Status));
54314| return Status;
54315| }
54316|
54317| //-----
    | -----
54318|
54319| ErrorCode
54320| PersistentDictionary::CloseAFile (
54321|     HANDLE      &FileHandle,
54322|     PFILE_OBJECT &FileObject,
54323|     HANDLE      &WaitHandle,
54324|     PVOID       &WaitObject )
54325| {
54326|     Profile("pd::CloseAFile");
54327|     Debug(DEBUG_DICT,("pd::CloseAFile %08x, %08x, %08x,
    | %08x\n",FileHandle,FileObject,WaitHandle,WaitObject));
54328|
54329|     ASSERT(IsValidHandle(FileHandle));
54330|     ASSERT(FileObject != NULL);
54331|
54332|     ZwClose(FileHandle);
54333|     FileHandle = INVALID_HANDLE_VALUE;
54334|     if ( FileObject ) {
54335|         ObDereferenceObject(FileObject);
54336|         FileObject = NULL;
54337|     }
54338|
54339|     ASSERT (IsValidHandle(WaitHandle));
54340|     ASSERT (WaitObject != NULL);
54341|
54342|     ZwClose (WaitHandle);

```

```

54343|   if ( WaitObject ) {
54344|       ObDereferenceObject (WaitObject);
54345|       WaitObject = NULL;
54346|   }
54347|
54348|   return 0;
54349| }
54350|
54351| //-----
54352| | -----
54353|
54354| ErrorCode
54355| PersistentDictionary::SetInfo(
54356| | tOpenTransactionInternal *In )
54357| {
54358|   if ( In ) {
54359|       // reflect changes back up to higher level code
54360|       In->QuiescentWait = QPeriod;
54361|       In->QuiescentTimeout = QTimeout;
54362|   }
54363|   return 0;
54364| }
54365| //-----
54366| | -----
54367| ULONG PersistentDictionary::QueryMaxNumUserSnapShots()
54368| {
54369|   GetRegistrySettings(&gRegistryPath);
54370|   return MaxNumUserSnapShots;
54371| }
54372| //-----
54373| | -----
54374| ErrorCode
54375| PersistentDictionary::InitClass(
54376|   ULONG          Stage,
54377|   tOpenTransactionInternal *In,
54378|   PVOID          AbortEvent )
54379| {
54380|   NTSTATUS Status=STATUS_UNSUCCESSFUL;
54381|   Debug(DEBUG_DICT,("pd::InitClass\n"));
54382|
54383|   if ( ClassInitd ) {
54384|       // already initd
54385|       return STATUS_SUCCESS;
54386|   }
54387|
54388|   DumpSizeOfs();

```

```

54389|
54390|  __try {
54391|      ULONG DefaultMemSize = 1;    // megabytes,
    | not nodes
54392|
54393|      ListHead = NULL;
54394|      SectorSize=512;
54395|      NumSavedHeaderBlocks =
    | DEFAULT_HEADER_BLOCKS_TO_SAVE;
54396|      NextHeaderBlockToSave = 0;
54397|
54398|
    | Reg_GetULONGKey(&gRegistryPath,L"FreeSpaceOptions",0,&PS
    | MFreeSpaceOptions);
54399|
    | Reg_GetULONGKey(&gRegistryPath,L"DirectIOOptions",0,&PSM
    | DirectIOOptions);
54400|      Debug(DEBUG_DICT,("pd::InitClass:
    | FreeSpaceOptions = %08x\n",PSMFreeSpaceOptions));
54401|      GetRegistrySettings(&gRegistryPath);
54402|
54403|      SetInfo(In);
54404|
54405|      if (
    | IncreaseNodeMemory(DefaultMemSize,1024*1024)==STATUS_SUC
    | CESS ) {
54406|          ClassInited = TRUE;
54407|          Status = STATUS_SUCCESS;
54408|      } else {
54409|          Debug(DEBUG_DICT,("Error! Out of memory for
    | nodelist\n"));
54410|          Status = STATUS_INSUFFICIENT_RESOURCES;
54411|      }
54412|  } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
54413|      Status = GetExceptionCode();
54414|      Debug(DEBUG_DICT,("Exception %08x in
    | pd::InitClass\n",Status));
54415|  }
54416|
54417|  return Status;
54418| }
54419|
54420| //-----
    | -----
54421|
54422| void CloseProcessSpecificHandles(
    | pCloseProcessSpecificHandles Psh )
54423| {
54424|

```

```

    | //Debug(DEBUG_DICT,("pd::CloseProcessSpecificHandles\n")
    | );
54425|
54426|     ASSERT (Psh != NULL);
54427|
54428|     if ( Psh != NULL ) {
54429|         if ( IsValidHandle(Psh->Handle) ) {
54430|             ZwClose(Psh->Handle);
54431|         }
54432|         Psh->Handle = INVALID_HANDLE_VALUE;
54433|
54434|         if ( Psh->Object != NULL ) {
54435|             ObDereferenceObject(Psh->Object);
54436|             Psh->Object = NULL;
54437|         }
54438|
54439|         pmSetEvent(&Psh->Event);
54440|     }
54441|
54442|     PsTerminateSystemThread( 0 );
54443| }
54444|
54445| //-----
    | -----
54446|
54447| void CloseProcessHandle ( HANDLE &h, PVOID &o )
54448| {
54449|     if(GlobalSystemProcessId == PsGetCurrentProcess())
    | {
54450|         if ( IsValidHandle(h) ) {
54451|             ZwClose(h);
54452|         }
54453|         h = INVALID_HANDLE_VALUE; // always change h
    | to INVALID_HANDLE_VALUE (it might have been NULL, for
    | example)
54454|         if ( o != NULL ) {
54455|             ObDereferenceObject(o);
54456|             o = NULL;
54457|         }
54458|     } else {
54459|         sCloseProcessSpecificHandles Psh (h, o);
    | // need constructor because it's only way to initialize
    | references in a struct
54460|         KeInitializeEvent (&Psh.Event,
    | NotificationEvent, NULL);
54461|         HANDLE ThreadHandle = INVALID_HANDLE_VALUE;
54462|         pmStartThread(
54463|
    | (PKSTART_ROUTINE)CloseProcessSpecificHandles,
54464|         (PVOID)&Psh,

```

```

54465|         &ThreadHandle );
54466|
54467|         if ( IsValidHandle(ThreadHandle) ) {
54468|             pmWaitForSingleObject(&Psh.Event,NULL);
54469|             ZwClose(ThreadHandle);
54470|         }
54471|     }
54472|
54473|     // Postconditions: handle and object must be
        | closed
54474|     ASSERT (h == INVALID_HANDLE_VALUE);
54475|     ASSERT (o == NULL);
54476| }
54477|
54478| //-----
        | -----
54479|
54480| ErrorCode
54481| PersistentDictionary::DeinitClass()
54482| {
54483|     Debug(DEBUG_DICT,("pd::DeinitClass\n"));
54484|
54485|     if ( GhostBusterTime ) {
54486|         return 0;
54487|     }
54488|
54489|     __try {
54490|         // InitClass always grab 1MB, lets free it.
54491|
54492|         ReclaimNodeMemory(1);
54493|
54494|     } __except(
        | ExceptionFilter(GetExceptionInformation()) ) {
54495|         Debug(DEBUG_DICT,("Exception %08x in
        | pd::DeinitClass\n",GetExceptionCode()));
54496|     }
54497|
54498|     ClassInitd = FALSE;
54499|     return 0;
54500| }
54501|
54502| //-----
        | -----
54503|
54504| PersistentDictionary::PersistentDictionary( )
54505| {
54506|     Debug(DEBUG_DICT,("pd::Constructor:
        | this=%08x\n",this));
54507|     initialize (DICT_FLAG_PERSISTENT);
54508| }

```

```

54509|
54510| //-----
    | -----
54511|
54512| PersistentDictionary::~PersistentDictionary( )
54513| {
54514|     Debug(DEBUG_DICT,("pd::Destructor:
    | this=%08x\n",this));
54515|     cleanup();
54516| }
54517|
54518| //-----
    | -----
54519|
54520| ErrorCode PersistentDictionary::initialize( ULONG
    | InitFlags )
54521| {
54522|     Debug(DEBUG_DICT,("pd::initialize:
    | this=%08x\n",this));
54523|     // zero out fields
54524|     Flags = InitFlags;
54525|     Next = ListHead;
54526|     ListHead = this;
54527|     SnapShot = NULL;
54528|
54529|     return STATUS_SUCCESS;
54530| }
54531|
54532| //-----
    | -----
54533|
54534| ErrorCode PersistentDictionary::GetDictionaryForVolume(
    | PVOID Volume, pDictionary &Dictionary )
54535| {
54536|     pPersistentDictionary p;
54537|     Dictionary = NULL;
54538|
54539|     p=ListHead;
54540|     while ( p ) {
54541|         if ( p->Volume == Volume ) {
54542|             Dictionary = p;
54543|             break;
54544|         }
54545|         p=p->Next;
54546|     }
54547|
54548|
54549|     return 0;
54550| }
54551|

```



```

54552| //-----
| -----
54553|
54554| ErrorCode PersistentDictionary::GetDictionaryForVolume
| (
54555|     PVOID        Volume,
54556|     pDictionary   &Dictionary,
54557|     ULONG         SequenceNumber )
54558| {
54559|     NTSTATUS Status = STATUS_NOT_FOUND;
54560|     Dictionary = NULL;
54561|
54562|     for ( pPersistentDictionary p = ListHead; p !=
| NULL; p = p->Next ) {
54563|         if ( (p->Volume == Volume) &&
| ((p->GetSequenceNumber()) == SequenceNumber) ) {
54564|             Status = STATUS_SUCCESS;
54565|             Dictionary = p;
54566|             break;
54567|         }
54568|     }
54569|
54570|     return Status;
54571| }
54572|
54573|
54574| //-----
| -----
54575|
54576| void PersistentDictionary::FreeNodeAndBits ( void *Ptr
| )
54577| {
54578|     Profile("pd::FreeNodeAndBits");
54579|     ASSERT(Ptr != NULL);
54580|
54581|     if ( Ptr != NULL ) {
54582|         pTreeLeaf Node = pTreeLeaf(Ptr);
54583|         if ( Node->Pos != INVALID_POSITION_VALUE ) {
54584|             ASSERT(Node->Pos <
| DevExt->Cache.PSManBitMapSize);
54585|             FreeCacheLocation (Node->Pos);
54586|         }
54587|
54588|         FreeNode(Node);
54589|     }
54590| }
54591|
54592| //-----
| -----
54593|

```

```

54594| ErrorCode PersistentDictionary::GetOutParams( WCHAR
      | *Cache )
54595| {
54596|     Debug(DEBUG_DICT,("pd::GetOutParams\n"));
54597|     return 0;
54598| }
54599|
54600| //-----
      | -----
54601|
54602| ErrorCode PersistentDictionary::GetVolumeSpaceUsed (
54603|     PDEVICE_OBJECT Volume,
54604|     ULONG          &At,
54605|     ULONG          &High,
54606|     ULONG          &Used )
54607| {
54608|     PFILTERED_EXTENSION DevExt =
      | GetFilteredExtension(Volume);
54609|     Debug(DEBUG_DICT,("pd::GetVolumeSpaceUsed\n"));
54610|
54611|     ASSERT (GRANULE_SIZE % 1024 == 0);
54612|     ASSERT (GRANULE_SIZE > 0);
54613|
54614|     At = DevExt->Cache.PSManBitMapSize * (GRANULE_SIZE
      | / 1024);
54615|     High = DevExt->Cache.PSManBitMapMaxSize *
      | (GRANULE_SIZE / 1024);
54616|     Used = DevExt->Cache.CurrentCacheFileSize *
      | (GRANULE_SIZE / 1024);
54617|
54618|     return 0;
54619| }
54620|
54621| //-----
      | -----
54622| // called when snapshot is destroyed via
      | Dictionary::Destroy
54623|
54624| ErrorCode PersistentDictionary::close()
54625| {
54626|     Debug(DEBUG_DICT,("pd::close\n"));
54627|     return STATUS_SUCCESS;
54628| }
54629|
54630| //-----
      | -----
54631|
54632| ErrorCode PersistentDictionary::destroy()
54633| {
54634|     NTSTATUS Status=STATUS_UNSUCCESSFUL;

```

```

54635|   Debug(DEBUG_DICT,("pd::destroy\n"));
54636|   __try {
54637|       cleanup();
54638|   //   Status = SbDeleteFile(CacheFileName);
54639|   } __except(
        | ExceptionFilter(GetExceptionInformation()) ) {
54640|       Status = GetExceptionCode();
54641|       Debug(DEBUG_DICT,("Exception %08x in
        | pd::destroy\n",Status));
54642|   }
54643|
54644|   return Status;
54645| }
54646|
54647| //-----
        | -----
54648|
54649| ErrorCode PersistentDictionary::reset()
54650| {
54651|   NTSTATUS Status = STATUS_SUCCESS;
54652|   Debug(DEBUG_DICT,("pd::reset\n"));
54653|   __try {
54654|       destroy();
54655|       Status = initialize(Flags);
54656|   } __except(
        | ExceptionFilter(GetExceptionInformation()) ) {
54657|       Status = GetExceptionCode();
54658|       Debug(DEBUG_DICT,("Exception %08x in
        | pd::reset\n",Status));
54659|   }
54660|   return Status;
54661| }
54662|
54663| //-----
        | -----
54664|
54665| BOOLEAN PersistentDictionary::DoFreeSpaceChecks()
54666| {
54667|   return (PSMFreeSpaceOptions &
        | PSM_DO_NOT_DO_FREE_SPACE) ? FALSE : TRUE;
54668| }
54669|
54670| //-----
        | -----
54671|
54672| void PersistentDictionary::DumpSizeOfs(void)
54673| {
54674|   | Debug(DEBUG_DICT,("Sizeof(PersistentDictionary)=%d\n",si
        | zeof(PersistentDictionary)));

```

```

54675|
    | Debug(DEBUG_DICT,("Sizeof(Dictionary)=%d\n",sizeof(Dicti
    | onary)));
54676|
    | Debug(DEBUG_DICT,("Sizeof(TemporaryDictionary)=%d\n",siz
    | eof(TemporaryDictionary)));
54677|
    | Debug(DEBUG_DICT,("Sizeof(tInternalSnapShot)=%d\n",sizeo
    | f(tInternalSnapShot)));
54678|
    | Debug(DEBUG_DICT,("Sizeof(tDiskInternalSnapShot)=%d\n",s
    | izeof(tDiskInternalSnapShot)));
54679|
54680|
    | Debug(DEBUG_DICT,("Sizeof(tRevertInfo)=%d\n",sizeof(tRev
    | ertInfo)));
54681|    Debug(DEBUG_DICT,(" tRevertInfo field
    | offsets:\n"));
54682|    Debug(DEBUG_DICT,("  SnapShotSequenceNumber
    | :%4d\n",
    | FIELD_OFFSET(tRevertInfo,SnapShotSequenceNumber) ));
54683|    Debug(DEBUG_DICT,("  Reserved2
    | :%4d\n", FIELD_OFFSET(tRevertInfo,Reserved2) ));
54684|    Debug(DEBUG_DICT,("  LastKnownGranuleFinished
    | :%4d\n",
    | FIELD_OFFSET(tRevertInfo,LastKnownGranuleFinished) ));
54685|    Debug(DEBUG_DICT,("  RevertFlags
    | :%4d\n", FIELD_OFFSET(tRevertInfo,RevertFlags) ));
54686|    Debug(DEBUG_DICT,("  Reserved3
    | :%4d\n", FIELD_OFFSET(tRevertInfo,Reserved3) ));
54687|    Debug(DEBUG_DICT,("  SnapShotTime
    | :%4d\n", FIELD_OFFSET(tRevertInfo,SnapShotTime) ));
54688|    Debug(DEBUG_DICT,("  Reserved[]
    | :%4d\n", FIELD_OFFSET(tRevertInfo,Reserved) ));
54689|
54690|
    | Debug(DEBUG_DICT,("Sizeof(tHeader)=%d\n",sizeof(tHeader)
    | ));
54691|
    | Debug(DEBUG_DICT,("Sizeof(tDiskNode)=%d\n",sizeof(tDiskN
    | ode)));
54692|
    | Debug(DEBUG_DICT,("Sizeof(tIndexSectorPrefix)=%d\n",size
    | of(tIndexSectorPrefix)));
54693|
    | Debug(DEBUG_DICT,("Sizeof(tIndexSectorInfo)=%d\n",sizeof
    | (tIndexSectorInfo)));
54694|
    | Debug(DEBUG_DICT,("Sizeof(tIndexSector)=%d\n",sizeof(tIn
    | dexSector)));

```

```

54695|
    | Debug(DEBUG_DICT,("Sizeof(tShared)=%d\n",sizeof(tShared)
    | ));
54696|
    | Debug(DEBUG_DICT,("Sizeof(tTreeLeaf)=%d\n",sizeof(tTreeL
    | eaf)));
54697|
    | Debug(DEBUG_DICT,("Sizeof(tTree)=%d\n",sizeof(tTree)));
54698|
    | Debug(DEBUG_DICT,("Sizeof(ZONE_SEGMENT_HEADER)=%d\n",siz
    | eof(ZONE_SEGMENT_HEADER)));
54699| }
54700|
54701| //-----
    | -----
54702|
54703| BOOLEAN PersistentDictionary::NeedsCaching(
    | ULARGE_INTEGER Sector, ULONG Count )
54704| {
54705|     LARGE_INTEGER Granule;
54706|     ULONG GranuleCount;
54707|     ULONG NumFree=0;
54708|     ULONG i;
54709|
54710|     Profile("pd::NeedsCaching");
54711|
54712|     if ( !Shared->Map ) {
54713|         return TRUE;
54714|     }
54715|
54716|     Granule.QuadPart = Sector.QuadPart /
    | SECTORS_PER_GRANULE;
54717|     GranuleCount=
    | (ULONG)((ROUND_UP(Sector.QuadPart+Count,SECTORS_PER_GRAN
    | ULE)-ROUND_DOWN(Sector.QuadPart,SECTORS_PER_GRANULE)) /
    | SECTORS_PER_GRANULE);
54718|
54719|     ASSERT(Granule.HighPart==0);
54720|
54721|     for ( i=0;i<GranuleCount;i++ ) {
54722|         PsmBitPositionValidate (Shared->Map,
    | Granule.LowPart + i);
54723|         if (
    | !RtlCheckBit(Shared->Map,Granule.LowPart+i) ) {
54724|             NumFree++;
54725|         } else {
54726|             break;
54727|         }
54728|     }
54729|

```

```

54730|   if ( NumFree==GranuleCount ) {
54731|       // all granules are free
54732|       return FALSE;
54733|   } else {
54734|       // one or more granules are used
54735|       return TRUE;
54736|   }
54737| }
54738|
54739|
54740| //-----
    | -----
54741|
54742|
54743| NTSTATUS PersistentDictionary::ProcessCachingMap(
    | PRTL_BITMAP *BitMapToWorkOn, const WCHAR
    | *VirtualVolName, const WCHAR *LiveVolName )
54744| {
54745|     ULONG ClusterSize=0;
54746|     NTSTATUS Status = FindAndProcessVirtualVolumeBitMap
    | (VirtualVolName, LiveVolName, BitMapToWorkOn,
    | ClusterSize );
54747|
54748| #if 0 // This restore has been moved farther up the
    | calling routine stack to shut loopholes where
54749|     // modifying a map being rebuilt could get into
    | live service.
54750|
54751|     // now that transformation is complete we can
    | reopen the map for freespace and cache control,
54752|     // instead of presuming everything should be tried
    | to be cached and letting duplicate tree hits control.
54753|     Shared->Map=Shared->MapInTransform;
54754|     Shared->MapInTransform = NULL;
54755| #endif
54756|
54757| #ifdef DEBUG
54758|     if(Shared->ClusterSize) {
54759|         // if reloading, make sure the cluster size
    | hasnt changed on us
54760|         // which would be catastrophic
54761|         if(ClusterSize) {
54762|             ASSERT(Shared->ClusterSize == ClusterSize);
54763|         }
54764|     }
54765| #endif
54766|
54767|     Shared->ClusterSize = ClusterSize;
54768|
54769|     return(Status);

```

```

54770| }
54771|
54772|
54773| NTSTATUS
    | PersistentDictionary::RemoveVirtualWrites(void)
54774| {
54775|     WCHAR Buffer2[257];
54776|     PDEVICE_OBJECT
        | Virtual=GetVdiskObjectForName(DevExt->Name,SnapShot->Per
        | manent.ExternalInstance);
54777|     NTSTATUS Status=STATUS_SUCCESS;
54778|
54779|     __try {
54780|         bool DismountedTheVirtualVolume = false;
54781|         PVDISK_EXTENSION VDisk = 0;
54782|         __try {
54783|             // we may not get an object if part2 has
            | not run yet
54784|             if(Virtual) {
54785|                 VDisk = GetVDiskExtension(Virtual);
54786|                 Debug(DEBUG_DICT,("RemoveVirtualWrites
            | called\n"));
54787|
            | swprintf(Buffer2,L"\\Device\\PsmDevices_%04x\\%s_%d",PSM
            | _LOW_COMPATIBLE_VERSION,VDisk->Name,SnapShot->Permanent.
            | ExternalInstance);
54788|                 Debug(DEBUG_DEVSUP,("Dismounting volume
            | '%S'\n",Buffer2));
54789|                 DismountedTheVirtualVolume = true;
54790|                 ASSERT(!VDisk->MountDisabled);
54791|                 VDisk->MountDisabled = 1;          //
            | don't allow mount while we are removing virtual writes
54792|                 Sblo_DismountVolume( Buffer2 );
54793|             }
54794|
54795|             Status = RemoveVirtualWritesFromTree
            | (TRUE);
54796|
54797|             if(Virtual) {
54798|                 Debug(DEBUG_DEVSUP,("Remounting
            | volume\n"));
54799|                 ASSERT(VDisk);
54800|                 VDisk->MountDisabled = 0;          // must
            | re-enable mount before we mount, don't you think?
54801|                 SbTouchVolume( Buffer2 );
54802|                 DismountedTheVirtualVolume = false;
54803|             }
54804|         } __finally {
54805|             if ( DismountedTheVirtualVolume ) {
54806|                 VDisk->MountDisabled = 0;          //

```

```

    | re-enable mount in case anything weird happens, so we
    | don't leave snapshot volume dead.
54807|     }
54808| }
54809| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
54810|     Status = GetExceptionCode();
54811|     Debug(DEBUG_DICT,("Exception %08x in
    | pd::RemoveVirtualWrites\n",Status));
54812| }
54813|
54814| return Status;
54815| }
54816|
54817| ULONG PersistentDictionary::GetSnapShotFlags(void)
54818| {
54819|     ULONG Flags = 0;
54820|     if ( SnapShot ) {
54821|         Flags = SnapShot->Permanent.SnapShotFlags;
54822|     } else {
54823|         Debug(DEBUG_DICT,("!!! pd::GetSnapShotFlags:
    | SnapShot==NULL !!!\n"));
54824|         ASSERT(FALSE);
54825|     }
54826|
54827|     return Flags;
54828| }
54829|
54830|
54831| ULONG PersistentDictionary::GetSequenceNumber() const
54832| {
54833|     ULONG sn = 0xffffffff;
54834|     if ( SnapShot ) {
54835|         sn = SnapShot->Permanent.SequenceNumber;
54836|     } else {
54837|         Debug(DEBUG_DICT,("!!! pd::GetSequenceNumber:
    | SnapShot==NULL !!!\n"));
54838|         ASSERT(FALSE);
54839|     }
54840|
54841| #ifdef DEBUG
54842|     /*
54843|     static const PersistentDictionary *
    | PreviousDictPrinted = 0;
54844|     static ULONG PreviousSnPrinted = 0;
54845|     static ULONG OmitCount = 0;
54846|
54847|     if ( this!=PreviousDictPrinted ||
    | sn!=PreviousSnPrinted ) { // cut down on the visual
    | noise

```



```

54848|     Debug(DEBUG_DICT,("pd::GetSequenceNumber:
| dict=%08x, returning %08x (did not print last %08x
| calls)\n",this,sn,OmitCount));
54849|     PreviousDictPrinted = (const
| PersistentDictionary *) this;
54850|     PreviousSnPrinted = sn;
54851|     OmitCount = 0;
54852| } else {
54853|     ++OmitCount;
54854| }
54855| */
54856| #endif /*DEBUG*/
54857|
54858| return sn;
54859| }
54860|
54861| __int64 PersistentDictionary::GetSnapShotTime() const
54862| {
54863|     __int64 time = __int64(0);
54864|
54865|     if ( SnapShot ) {
54866|         time =
| SnapShot->Permanent.SnapShotTime.QuadPart;
54867|     } else {
54868|         Debug(DEBUG_DICT,("!!! pd::GetSnapShotTime:
| SnapShot==NULL !!!\n"));
54869|         ASSERT(FALSE);
54870|     }
54871|
54872|     return time;
54873| }
54874|
54875| NTSTATUS PersistentDictionary::SetRevertBootInfo (
54876|     PDEVICE_OBJECT Volume,
54877|     const tRevertInfo &RevertInfo )
54878| {
54879|     NTSTATUS Status = STATUS_SUCCESS;
54880|     Debug(DEBUG_DICT,("pd::SetRevertBootInfo:
| Volume=%08x\n",Volume));
54881|
54882|     PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
54883|     if ( DevExt->Cache.Header ) {
54884|         DevExt->Cache.Header->RevertInfo = RevertInfo;
54885|
54886|         Debug(DEBUG_DEVSUP,("SetRevertBootInfo:\n"));
54887|         Debug(DEBUG_DEVSUP,(" SnapShotSequenceNumber
| = %08x\n",RevertInfo.SnapShotSequenceNumber));
54888|         Debug(DEBUG_DEVSUP,("
| LastKnownGranuleFinished =

```

```

    | %016l64x\n",RevertInfo.LastKnownGranuleFinished.QuadPart
    | ));
54889|     } else {
54890|         Debug(DEBUG_DEVSUP,("SetRevertBootInfo: Header
    | data does not exist!\n"));
54891|     }
54892|
54893|
54894|     Debug(DEBUG_DICT,("pd::SetRevertBootInfo(%08x)
    | returning %08x\n",Volume,Status));
54895|     return Status;
54896| }
54897|
54898|
54899| void PersistentDictionary::GetCacheThresholds(
    | PDEVICE_OBJECT Volume, ULONG &Warn, ULONG &Full, ULONG
    | &Interval, ULONG &FullPercent, ULONG &FullAction )
54900| {
54901|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
54902|     Warn = DevExt->Cache.CacheWarningThresholdPercent;
54903|     Full = DevExt->Cache.CacheFullThresholdPercent;
54904|     Interval = DevExt->Cache.CacheWarningInterval;
54905|     FullPercent = DevExt->Cache.CacheFullActionPercent;
54906|     FullAction = DevExt->Cache.CacheFullAction;
54907|     return;
54908| }
54909|
54910|
54911| NTSTATUS PersistentDictionary::LoadSnapShotsForVolume(
    | PDEVICE_OBJECT Volume,BOOLEAN Rebuild, PVOID AbortEvent
    | )
54912| {
54913|     NTSTATUS Status = STATUS_SUCCESS;
54914|     ULONG Opened=FALSE;
54915|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
54916|
54917|     Debug(DEBUG_DICT,("pd::LoadSnapShotsForVolume:
    | Volume=%08x, Rebuild=%s, IsMounted=%s\n",
54918|         Volume,
54919|         (Rebuild ? "TRUE" : "FALSE"),
54920|         (DevExt->IsMounted ? "TRUE" : "FALSE") ));
54921|
54922|     if(!DevExt->OpenCloseAcquired) {
54923|         if (
    | AcquireOpenCloseResourceOnly((PKEVENT)AbortEvent)!=STATU
    | S_WAIT_0 ) {
54924|             return PSM_CANCELED_BY_USER;
54925|         }

```

```

54926|     Opened = TRUE;
54927|     DevExt->OpenCloseAcquired = TRUE;
54928| }
54929|
54930| __try {
54931| #if 0
54932|     // we need to "refresh" what we know about the
        | volumes
54933|     if(DevExt->PSMed) {
54934|         // snapshots already loaded, this can
        | happen when the volume comes online and
54935|         // when the volume is mounted, which
        | appears to happen in 2 seperate threads
54936|         Debug(DEBUG_DICT,("LoadSnapShotsForVolume:
        | Snapshots are already loaded for this volume
        | %08x\n",Volume));
54937|         try_return(Status=STATUS_SUCCESS);
54938|     }
54939| #endif
54940|
54941|     // set up system process as all snapshots must
        | be assigned to a process
54942|     pOT_USER User =
        | FindPSMUser(GlobalSystemProcessId,(_ETHREAD*)-2);
54943|     ASSERT(User);
54944|
54945|     if ( User ) {
54946|         ULONG lInited = FALSE;
54947|         User->Persistent = TRUE;
54948|         User->SaveTempOnExit = FALSE;
54949|
54950|         // dont do this if already done
54951|         if (
            | IsValidHandle(DevExt->Cache.CacheFile.FileHandle) ) {
54952|             try_return(Status=STATUS_SUCCESS);
54953|         }
54954|
54955|         DevExt->InLoadUnload = TRUE;
54956|
54957|         if ( !PSManPSMlInited ) {
54958|             lInited = TRUE;
54959|
            | UpdateGlobalStatus(PSM_RESOURCE_ACQUISITION);
54960|             Status = SbPreInit(User,NULL);
54961|         }
54962|
54963|         if ( NT_SUCCESS(Status) ) {
54964|             Status =
            | RebuildSnapShotsForVolume(Volume,Rebuild,AbortEvent);
54965|             if ( NT_SUCCESS(Status) ) {

```

```

54966|         try_return(NOTHING);
54967|     } else {
54968|         Debug(DEBUG_DICT,("Error %08x
| rebuilding snapshots\n",Status));
54969|     }
54970|
54971|     if ( !Initd ) {
54972|         SbPreDelInit();
54973|     }
54974|     } else { // if SbPreInit
54975|         Debug(DEBUG_DICT,("Error %08x initing
| snapshot engine\n",Status));
54976|     }
54977|     } else { // if user
54978|         Status = STATUS_INSUFFICIENT_RESOURCES;
54979|     }
54980| try_exit: NOTHING;
54981| } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
54982|     Status = GetExceptionCode();
54983|     Debug(DEBUG_DICT,("Exception %08x in
| pd::LoadSnapShotsForVolume\n",Status));
54984| }
54985| UpdateGlobalStatus(PSM_IDLE);
54986| if(Opened) {
54987|     DevExt->OpenCloseAcquired = FALSE;
54988|     ReleaseOpenCloseResource();
54989| }
54990| pmThreadSwitch();
54991| DevExt->InLoadUnload = FALSE;
54992| Debug(DEBUG_DICT,("pd::LoadSnapShotsForVolume
| returning %08x\n",Status));
54993| return Status;
54994| }
54995|
54996|
54997| NTSTATUS RemoveVolumesFromSystem( PDEVICE_OBJECT
| Volume)
54998| {
54999|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
55000|     PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
55001|
55002|     Debug(DEBUG_DICT,("Entering
| RemoveVolumesFromSystem, Vol=%08x\n",Volume));
55003|
55004|     __try {
55005|         pOT_USER User =
| FindPSMUser(GlobalSystemProcessId,(_ETHREAD*)-2);
55006|         ASSERT(User);

```

```

55007|
55008|    // remove all virtual volumes
55009|    GetSnapShotForWrite();
55010|    __try {
55011|        PLIST_ENTRY
        | ListEntry=DevExt->SnapShots.Flink;
55012|        while(ListEntry != &DevExt->SnapShots) {
55013|            pkSnapShotEntry
        | p=CONTAINING_RECORD(ListEntry,tkSnapShotEntry,DevExt);
55014|
55015|            ASSERT(p->DeviceObject == Volume);
55016|
55017|            UseSnapShot(p);
55018|
55019|            // start back at top
55020|
        | Debug(DEBUG_DCPSM,("RemoveVolumesFromSystem: Freeing
        | device %08x!\n",p->DeviceObject));
55021|            Status =
        | FreePSMVolume(User,p->DeviceObject,p);
55022|
55023|            // assert that no other people have
        | this snapshot in use
55024|            ASSERT(p->Count==1);
55025|            pkSnapShotMaster
        | Master=p->MasterSnapShot;
55026|
55027|            // get rid of last reference count so
        | the snapshot is deleted
55028|            DoneWithSnapShot(p);
55029|
55030|
        | Debug(DEBUG_DICT,("pd::RemoveSnapShotsForVolume:
        | snapshot entry deleted, freeing master
        | %08x!\n",Master->Count));
55031|            if(Master->Count==0) {
55032|                FREE_POINTER(Master);
55033|                ASSERT(GlobalData->NumActive);
55034|                InterlockedDecrement((PLONG)
        | &GlobalData->NumActive);
55035|                InterlockedDecrement((PLONG)
        | &User->Open);
55036|
        | Debug(DEBUG_DICT,("pd::RemoveSnapShotsForVolume:
        | Decrementd NumActive to %08x
        | open=%08x!\n",GlobalData->NumActive,User->Open));
55037|            }
55038|
55039|            ListEntry = DevExt->SnapShots.Flink;
55040|        }

```

```

55041|     } __finally {
55042|         ReleaseSnapShotForWrite();
55043|     }
55044| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
55045|     Status = GetExceptionCode();
55046|     Debug(DEBUG_SFILTER,("Exception %08x in
    | dismount all volumes\n",Status));
55047| }
55048|
55049| Debug(DEBUG_DICT,("RemoveVolumesFromSystem
    | returning %08x\n",Status));
55050| return Status;
55051| }
55052|
55053| // Undoes what TdAddDrive did.
55054| NTSTATUS DeleteVirtualVolumesFromSystem( PDEVICE_OBJECT
    | Volume )
55055| {
55056|     NTSTATUS Status=STATUS_SUCCESS;
55057|     Debug(DEBUG_DICT,("Entering
    | DeleteVirtualVolumesFromSystem, Vol=%08x\n",Volume));
55058|
55059|     __try {
55060|         PDEVICE_OBJECT DevObj =
    | PSMAN_DRIVER_OBJECT->DeviceObject;
55061|         while(DevObj) {
55062|             if (
    | PsmGetObjectTypeInfo(DevObj)==OBJECT_VIRTUALDISK ) {
55063|                 PVDISK_EXTENSION DevExt =
    | GetVDiskExtension(DevObj);
55064|                 Debug(DEBUG_VDISK,("VDisk:
    | DeleteVirtualVolumesFromSystem: Found virtual Volume
    | %08x\n",DevObj));
55065|                 if(DevExt->PSMDevice==Volume) {
55066|                     Debug(DEBUG_VDISK,("VDisk:
    | DeleteVirtualVolumesFromSystem: Found virtual Volume
    | %08x to delete\n",DevObj));
55067|                     if ((DevExt->DeviceObject->Vpb) &&
    | ( DevExt->DeviceObject->Vpb->Flags & VPB_MOUNTED )) {
55068|                         Debug(DEBUG_VDISK,("VDisk:
    | Devcon: Volume %08x is mounted\n",DevObj));
55069|                         DevExt->DeviceObject->Flags |=
    | DO_VERIFY_VOLUME;
55070|                     }
55071|
55072|                     DevExt->PartitionActive = FALSE;
55073|                     DevExt->DriveNotReady = TRUE;
55074|                     DevExt->PSMDevice = NULL;
55075|                     DevExt->MountDisabled=TRUE;

```

```

55076|
55077|         UNICODE_STRING Uni;
55078|
55079|         RtlInitUnicodeString(&Uni,
    | DevExt->VolumeGuid);
55080|         IoDeleteSymbolicLink(&Uni);
55081|
55082|         DoneWithSnapShot(DevExt->SnapShot);
55083|         DevExt->SnapShot = NULL;
55084|         DevExt->MasterSnapShot=NULL;
55085|     }
55086| }
55087|     DevObj=DevObj->NextDevice;
55088| }
55089| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
55090|     Status = GetExceptionCode();
55091|     Debug(DEBUG_SFILTER,("Exception %08x in
    | DeleteVirtualVolumesFromSystem\n",Status));
55092| }
55093|
55094|     Debug(DEBUG_DICT,("DeleteVirtualVolumesFromSystem
    | returning %08x\n",Status));
55095|     return Status;
55096| }
55097|
55098| typedef struct sClusterFiles {
55099|     WCHAR *KeyName;
55100|     PFILE_OBJECT FileObject;
55101| } tClusterFiles, *pClusterFiles;
55102|
55103| #ifdef DEBUG
55104| void DumpMap( RETRIEVAL_POINTERS_BUFFER *Map )
55105| {
55106|     if(Map) {
55107|         ULONG j;
55108|         LARGE_INTEGER StartVcn;
55109|
55110|         StartVcn = Map->StartingVcn;
55111|         for(j=0;j<Map->ExtentCount;j++) {
55112|             Debug(DEBUG_DICT,(" %04d StartVcn=%08l64x,
    | LCN=%08l64x, Len=%08l64x, Next=%08l64x\n",j,
55113|                 StartVcn.QuadPart,
55114|                 Map->Extents[j].Lcn.QuadPart,
55115|                 Map->Extents[j].NextVcn.QuadPart-StartVcn.QuadPart,
55116|                 Map->Extents[j].NextVcn.QuadPart));
55117|             StartVcn = Map->Extents[j].NextVcn;
55118|         }
55119|     }

```

```

55120| }
55121| #endif
55122|
55123| NTSTATUS PersistentDictionary::RetrieveDirectIOMaps(
    | PDEVICE_OBJECT Volume, BOOLEAN CheckClusterDataBase)
55124| {
55125|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
55126|     WCHAR *ClusterReg;
55127|     WCHAR *LocalReg={0};
55128|     UNICODE_STRING Uni;
55129|     RETRIEVAL_POINTERS_BUFFER *RP = NULL;
55130|     ULONG BPC,BPS;
55131|     PVOID Handle;
55132|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
55133|     DirectAccessFile *newHeaderDirect = new
    | DirectAccessFile();
55134|     DirectAccessFile *newIndexDirect = new
    | DirectAccessFile();
55135|     DirectAccessFile *newCacheDirect = new
    | DirectAccessFile();
55136|
55137|     Profile("pd::RetrieveDirectIOMaps");
55138|
55139|     Debug(DEBUG_DICT,("RetrieveDirectIOMaps %08x: %08x
    | %08x %08x\n",Volume,
    | DevExt->Cache.HeaderFile.Direct,DevExt->Cache.IndexFile.
    | Direct,DevExt->Cache.CacheFile.Direct));
55140|     LocalReg=(WCHAR*)MemAllocateString(256);
55141|     if(LocalReg) {
55142|
    | RtlCopyMemory(LocalReg,gRegistryPath.Buffer,gRegistryPat
    | h.Length);
55143|     LocalReg[gRegistryPath.Length / 2] = 0;
55144|     wcscat(LocalReg,L"\");
55145|     wcscat(LocalReg,DevExt->VolumeGuid);
55146|
55147|     RtlInitUnicodeString(&Uni,LocalReg);
55148|     ClusterReg=(WCHAR*)MemAllocateString(256);
55149|     if(ClusterReg) {
55150|         ULONG DidCluster = TRUE;
55151|
55152|
    | wcscpy(ClusterReg,L"\\Registry\\Machine\\Cluster\\Persis
    | tentStorageManager\\");
55153|         wcscat(ClusterReg,DevExt->UniqueId);
55154|         if(CheckClusterDataBase) {
55155|             // if cluster entry exists, use it
    | instead, as the local copy may be stale
55156|

```



```

    | if(RtlCheckRegistryKey(RTL_REGISTRY_ABSOLUTE,ClusterReg)
    | ==STATUS_SUCCESS) {
55157|
    | Debug(DEBUG_DICT,("pd:RetrieveDirectIOMaps: Using
    | cluster database instead of local database\n"));
55158|
55159|         // copy cluster database to local
    | database so it doesnt get stale
55160|
    | CopyClusterRegistryToLocalRegistry(Volume,&Uni);
55161|
55162|
    | RtlInitUnicodeString(&Uni,ClusterReg);
55163|         DidCluster = FALSE;
55164|     }
55165| }
55166|
55167|     // FIXFIXFIX need to direct the direct io
    | stuff to the volume that
55168|     // contains the file, not the volume we are
    | reading( which may not be the same!)
55169|
55170|     // FIXFIXFIX this is per file, not global
    | for all files. (as they may be
55171|     // on different volumes)
55172|
    | Reg_GetULONGKey(&Uni,L"BytesPerCluster",4096,&BPC);
55173|
    | Reg_GetULONGKey(&Uni,L"BytesPerSector",512,&BPS);
55174|
55175| DoAgain:
55176|     Status =
    | Reg_GetBinaryKey(&Uni,L"HeaderMap",(PVOID*)&RP,&Handle);
55177|     if ( NT_SUCCESS(Status) ) {
55178|
    | newHeaderDirect->SetInternal(Volume,NULL,RP,BPC,BPS);
55179|     #ifdef DEBUG
55180|         Debug(DEBUG_DICT,("Header Map:\n"));
55181|         DumpMap(RP);
55182|     #endif
55183|         Reg_FreeBinary(Handle);
55184|     } else {
55185|
    | Debug(DEBUG_DICT,("pd:RetrieveDirectIOMaps: Could not
    | get HeaderMap from registry (%08x)\n",Status));
55186|         // ok, key not there.. lets try local
55187|         if(!DidCluster) {
55188|
    | Debug(DEBUG_DICT,("pd:RetrieveDirectIOMaps: Using local
    | database instead of cluster database\n"));

```

```

55189|         DidCluster=TRUE;
55190|
55191|         | RtlInitUnicodeString(&Uni,LocalReg);
55192|         goto DoAgain;
55193|     }
55194|
55195|     if(NT_SUCCESS(Status)) {
55196|         RP = NULL;
55197|         Status =
55198|         | Reg_GetBinaryKey(&Uni,L"IndexMap",(PVOID*)&RP,&Handle);
55199|         if ( NT_SUCCESS(Status) ) {
55200|
55201|         | newIndexDirect->SetInternal(Volume,NULL,RP,BPC,BPS);
55202|         #ifdef DEBUG
55203|         | Debug(DEBUG_DICT,("Index Map:\n"));
55204|         | DumpMap(RP);
55205|         #endif
55206|         | Reg_FreeBinary(Handle);
55207|         } else {
55208|
55209|         | Debug(DEBUG_DICT,("pd::RetrieveDirectIOMaps: Could not
55210|         | get IndexMap from registry (%08x)\n",Status));
55211|         }
55212|
55213|         if(NT_SUCCESS(Status)) {
55214|             RP = NULL;
55215|             Status =
55216|             | Reg_GetBinaryKey(&Uni,L"CacheMap",(PVOID*)&RP,&Handle);
55217|             if ( NT_SUCCESS(Status) ) {
55218|
55219|             | newCacheDirect->SetInternal(Volume,NULL,RP,BPC,BPS);
55220|             #ifdef DEBUG
55221|             | Debug(DEBUG_DICT,("Cache
55222|             | Map:\n"));
55223|             | DumpMap(RP);
55224|             #endif
55225|             | Reg_FreeBinary(Handle);
55226|             } else {
55227|
55228|             | Debug(DEBUG_DICT,("pd::RetrieveDirectIOMaps: Could not
55229|             | get CacheMap from registry (%08x)\n",Status));
55230|             }
55231|             RP = NULL;
55232|         }
55233|     }
55234|     MemFreeString(ClusterReg);
55235| } else {
55236|     Status = STATUS_INSUFFICIENT_RESOURCES;
55237| }

```

```

    | Debug(DEBUG_DICT,("pd::RetrieveDirectIOMaps: Out of
    | memory for clusterreg\n"));
55229|     }
55230|     MemFreeString(LocalReg);
55231| } else {
55232|     Status = STATUS_INSUFFICIENT_RESOURCES;
55233|     Debug(DEBUG_DICT,("pd::RetrieveDirectIOMaps:
    | Out of memory for localreg\n"));
55234| }
55235|
55236| if ( !NT_SUCCESS(Status) ) {
55237|     Debug(DEBUG_DICT,("pd::RetrieveDirectIOMaps:
    | Deleting new maps because of status %08x\n",Status));
55238|     delete newHeaderDirect;
55239|     delete newIndexDirect;
55240|     delete newCacheDirect;
55241|
55242|     newHeaderDirect = 0;
55243|     newIndexDirect = 0;
55244|     newCacheDirect = 0;
55245| }
55246|
55247| Debug(DEBUG_DICT,("pd::RetrieveDirectIOMaps: About
    | to acquire writer lock for DirectFileAccess pointer
    | swap\n"));
55248| KeEnterCriticalRegion();
55249| pmAcquireWriterLock (
    | &DevExt->Cache.DirectAccessResource, TRUE );
55250| DirectAccessFile *oldHeaderDirect =
    | DevExt->Cache.HeaderFile.Direct;
55251| DirectAccessFile *oldIndexDirect =
    | DevExt->Cache.IndexFile.Direct;
55252| DirectAccessFile *oldCacheDirect =
    | DevExt->Cache.CacheFile.Direct;
55253| DevExt->Cache.HeaderFile.Direct = newHeaderDirect;
55254| DevExt->Cache.IndexFile.Direct = newIndexDirect;
55255| DevExt->Cache.CacheFile.Direct = newCacheDirect;
55256| pmReleaseWriterLock (
    | &DevExt->Cache.DirectAccessResource );
55257| KeLeaveCriticalRegion();
55258| Debug(DEBUG_DICT,("pd::RetrieveDirectIOMaps:
    | Released writer lock - header=%08x, index=%08x,
    | cache=%08x\n",
55259|     DevExt->Cache.HeaderFile.Direct,
55260|     DevExt->Cache.IndexFile.Direct,
55261|     DevExt->Cache.CacheFile.Direct));
55262|
55263| Debug(DEBUG_DICT,("pd::RetrieveDirectIOMaps:
    | deleting old DirectAccessFile objects: header=%08x,
    | index=%08x,

```

```

    | cache=%08x\n",oldHeaderDirect,oldIndexDirect,oldCacheDir
    | ect));
55264|   delete oldHeaderDirect;
55265|   delete oldIndexDirect;
55266|   delete oldCacheDirect;
55267|
55268|   return Status;
55269| }
55270|
55271| NTSTATUS PersistentDictionary::StoreClustersOfFiles(
    | PDEVICE_OBJECT Volume )
55272| {
55273|   NTSTATUS Status=STATUS_UNSUCCESSFUL;
55274|   WCHAR Buffer[255]={0};
55275|   ULONG i;
55276|   PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
55277|
55278|   Debug(DEBUG_DICT,("StoreClustersOfFiles:
    | Volume=%08x\n",Volume));
55279|   Profile("pd::StoreClustersOfFiles");
55280|
55281| #ifdef DEBUG
55282|   if(IsSnapShotAcquiredForWrite()) {
55283|       Debug(DEBUG_DCPSM,("TdAddDrive: Snapshot
    | acquired for write!!! This will cause a deadlock\n"));
55284|       DbgBreakPoint();
55285|   }
55286| #endif
55287|
55288|   __try {
55289| #define FileCount 3
55290|       tClusterFiles Files[FileCount]={
55291|
    | {L"HeaderMap",DevExt->Cache.HeaderFile.FileObject},
55292|
    | {L"IndexMap",DevExt->Cache.IndexFile.FileObject},
55293|
    | {L"CacheMap",DevExt->Cache.CacheFile.FileObject}
55294|       };
55295|
55296|
    | RtlCopyMemory(Buffer,gRegistryPath.Buffer,gRegistryPath.
    | Length);
55297|   Buffer[gRegistryPath.Length / 2] = 0;
55298|   wcscat(Buffer,L"\n");
55299|   wcscat(Buffer,DevExt->VolumeGuid);
55300|
55301|   for(i=0;i<FileCount;i++) {
55302|       RETRIEVAL_POINTERS_BUFFER *RP;

```

```

55303|     STARTING_VCN_INPUT_BUFFER SVIB;
55304|     ULONG CalcSize=0;
55305|
55306|     if(!IsValidHandle(Files[i].FileObject)) {
55307|         continue;
55308|     }
55309|     ULONG Count = 16;
55310|     RP=NULL;
55311|
55312|     do {
55313|         if(RP) {
55314|             MemFreePool(RP);
55315|             RP = NULL;
55316|         }
55317|         CalcSize =
55318|         | FIELD_OFFSET(RETRIEVAL_POINTERS_BUFFER,Extents)+Count*si
55319|         | sizeof(RP->Extents);
55320|         RP =
55321|         | (RETRIEVAL_POINTERS_BUFFER*)MemAllocatePoolWithTag(Paged
55322|         | Pool,CalcSize,TEMPTAG);
55323|         if(RP) {
55324|             RtlZeroMemory(RP,CalcSize);
55325|             SVIB.StartingVcn.QuadPart=0;
55326|             Status = FS_GetFileMap(
55327|             | Files[i].FileObject,
55328|             | &SVIB,
55329|             | RP,
55330|             | CalcSize);
55331|             Count*=2;
55332|         } else {
55333|             Status =
55334|             | STATUS_INSUFFICIENT_RESOURCES;
55335|         }
55336|     } while(Status == STATUS_BUFFER_OVERFLOW);
55337|
55338|     if(Status==STATUS_SUCCESS) {
55339|         LARGE_INTEGER Total,Avail;
55340|         ULONG BPS,BPC;
55341|
55342|         | FS_GetVolumeInfo(Files[i].FileObject,BPC,BPS,Total,Avail
55343|         | );
55344|
55345|         | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Byt
55346|         | esPerCluster",REG_DWORD,&BPC,sizeof(DWORD));
55347|
55348|         | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Byt
55349|         | esPerSector",REG_DWORD,&BPS,sizeof(DWORD));

```

```

55342|
55343| #ifdef DEBUG
55344|
55345|     | Debug(DEBUG_DICT,("%S\n",Files[i].KeyName));
55346|     DumpMap(RP);
55347| #endif
55348|     Status = RtlWriteRegistryValue(
55349|         RTL_REGISTRY_ABSOLUTE,
55350|         Buffer,
55351|         Files[i].KeyName,
55352|         REG_BINARY,
55353|         RP,
55354|         CalcSize
55355|     );
55356|     if(!NT_SUCCESS(Status)) {
55357|
55358|         | Debug(DEBUG_DEVCON,("pd::StoreClustersOfFiles: Error
55359|         | %08x setting file map in reg\n",Status));
55360|     }
55361|     MemFreePool(RP);
55362|     } else {
55363|
55364|         | Debug(DEBUG_DEVCON,("pd::StoreClustersOfFiles: Error
55365|         | %08x getting file map size\n",Status));
55366|     }
55367|     } __except(
55368|         | ExceptionFilter(GetExceptionInformation()) ) {
55369|         Status = GetExceptionCode();
55370|         Debug(DEBUG_SFILTER,("Exception %08x in
55371|         | StoreClustersOfFiles\n",Status));
55372|     }
55373|     Debug(DEBUG_DICT,("StoreClustersOfFiles: returning
55374|     | %08x\n",Status));
55375|     return STATUS_SUCCESS;
55376| }
55377|
55378| /*
55379| Purpose of this function is to remove all snapshots
55380| on the internal
55381| list that did not get freed when the volumes were
55382| freed. This can
55383| happen when virtual volumes have not been mapped in
55384| yet, as most
55385| code following the virtual device tree
55386|
55387| */

```

```

55381|
55382| NTSTATUS PersistentDictionary::RemoveSnapShotDroppings(
    | PDEVICE_OBJECT Volume )
55383| {
55384|     PFILTERED_EXTENSION DevExt =
        | GetFilteredExtension(Volume);
55385|     // get system user
55386|     pOT_USER User =
        | FindPSMUser(GlobalSystemProcessId,(_ETHREAD*)-2);
55387|     PLIST_ENTRY ListEntry;
55388|
55389|     Debug(DEBUG_DICT,("RemoveSnapShotDroppings: Called
        | for volume %08x\n",Volume));
55390|     GetSnapShotForWrite();
55391|     __try {
55392|         ListEntry=DevExt->Cache.SnapShotHead.Flink;
55393|         while ( ListEntry!=&DevExt->Cache.SnapShotHead
            | ) {
55394|             pInternalSnapShot
            | s=CONTAINING_RECORD(ListEntry,tInternalSnapShot,ListEntr
            | y);
55395|             Debug(DEBUG_DICT,("RemoveSnapShotDroppings:
            | Found internal ss %08x, rc=%d,
            | master=%08x\n",s,s->ReferenceCount,s->SnapShotMaster));
55396|
55397|             // go through list of all snapshot entries
55398|             pkSnapShotEntry
            | p=GetTopSnapShotForMaster(&s->SnapShotMaster->SnapShots)
            | ;
55399|             while ( p ) {
55400|
            | Debug(DEBUG_DICT,("RemoveSnapShotDroppings: Found ss
            | entry %08x, %08x\n",p,p->DeviceObject));
55401|             // and delete ones for this volume
55402|             if (p->DeviceObject==Volume) {
55403|
            | Debug(DEBUG_DCPSM,("RemoveSnapShotDroppings: Deleting
            | dropping %08x!\n",p));
55404|
55405|             if (
            | DelDeviceFromList(&User->SnapShots,p)==STATUS_SUCCESS )
            | {
55406|                 if ( User->NumOpenSnapShots ) {
55407|
            | InterlockedDecrement((PLONG) &User->NumOpenSnapShots);
55408|                 } else {
55409|                     // FIXFIXFIX can this
            | happen?
55410|
            | Debug(DEBUG_DCPSM,("RemoveSnapShotDroppings: Doesnt

```

```

    | have psm active for that volume!\n"));
55411|     #ifdef DEBUG
55412|         DbgBreakPoint();
55413|     #endif
55414|     }
55415| } else {
55416|     // This happens when virtual
    | volumes havent been mapped in yet
55417|
    | Debug(DEBUG_DCPSM,("RemoveSnapShotDroppings: Unable to
    | delete from list!\n"));
55418|     }
55419|
55420|     // this deletes the snapshot and
    | removes it from the internal lists
55421|     FreeResourcesForVolume(Volume,p);
55422|
55423|     ASSERT(p->Count>1);
55424|
55425|     ULONG Deleted = p->Count==1;
55426|     pkSnapShotMaster
    | Master=p->MasterSnapShot;
55427|
55428|     // get rid of last reference count
    | so the snapshot is deleted
55429|     DoneWithSnapShot(p);
55430|
55431|     ASSERT(Deleted);
55432|
55433|     // if no more references to this
    | snapshot entry then
55434|     // get rid of reference count
55435|     if(Deleted) {
55436|
    | Debug(DEBUG_DICT,("pd::RemoveSnapShotsForVolume:
    | snapshot entry deleting, decrementing master
    | %08x\n",Master->Count));
55437|         InterlockedDecrement((PLONG)
    | &Master->Count);
55438|         if(Master->Count==0) {
55439|             FREE_POINTER(Master);
55440|
    | ASSERT(GlobalData->NumActive);
55441|
    | InterlockedDecrement((PLONG) &GlobalData->NumActive);
55442|
    | InterlockedDecrement((PLONG) &User->Open);
55443|
    | Debug(DEBUG_DICT,("pd::RemoveSnapShotsForVolume:
    | Decrementd NumActive to %08x

```



```

    | open=%08x\n",GlobalData->NumActive,User->Open));
55444|         }
55445|     }
55446|
55447|         // start back at top
55448|
    | p=GetTopSnapShotForMaster(&s->SnapShotMaster->SnapShots)
    | ;
55449|     } else {
55450|
    | Debug(DEBUG_DICT,("RemoveSnapShotDroppings: Looking for
    | %08x, but it is %08x\n",Volume,p->DeviceObject));
55451|
    | p=GetNextSnapShotForMaster(&s->SnapShotMaster->SnapShots
    | ,p);
55452|     }
55453| }
55454|
55455|     ListEntry=ListEntry->Flink;
55456| }
55457| } __finally {
55458|     ReleaseSnapShotForWrite();
55459| }
55460| Debug(DEBUG_DICT,("RemoveSnapShotDroppings:
    | Exiting\n"));
55461| return STATUS_SUCCESS;
55462| }
55463|
55464| NTSTATUS
    | PersistentDictionary::UnloadSnapShotsForVolume(
    | PDEVICE_OBJECT Volume, ULONG OpenCloseOwned )
55465| {
55466|     NTSTATUS Status=STATUS_SUCCESS;
55467| #if 1
55468|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
55469|
55470|     // this is in the context of the calling thread.
    | IE autochk or mount.exe, etc..
55471|     // do not assume GlobalSystemProcessId ==
    | PsGetCurrentProcess()
55472|
55473|     Debug(DEBUG_DICT,("pd::UnloadSnapShotsForVolume:
    | Volume=%08x, '%S'\n",Volume,DevExt->Name));
55474|
55475|     if(!OpenCloseOwned) {
55476|         // cancel all pending creates
55477|         CancelCreationOfSnapShots();
55478|
55479|         if(DevExt->OpenCloseAcquired) {

```

```

55480|          // this happens when we are called
| recursively
55481|          // A. OpenClose resource is acquired
| (because snapshot is being created)
55482|          // B. We think a volume is mounted (on
| cluster, because we were
55483|          // not notified it was removed)
55484|          // C. We call ZwCreateFile to open file
55485|          // D. It initiates a mount when it realizes
| the volume isnt mounted
55486|          // E. It then calls rebuild
55487|          // F. Which then calls unload so it can
| reload the snapshots
55488|          // G. Attempt to acquire open close
| resource again.
55489|
55490|          // so we dont try and close it
55491|          OpenCloseOwned = TRUE;
55492|      } else {
55493|          if (
| AcquireOpenCloseResourceOnly(NULL)!=STATUS_WAIT_0 ) {
55494|              return PSM_CANCELED_BY_USER;
55495|          }
55496|      }
55497|  }
55498|
55499|  __try {
55500|
55501|      UpdateGlobalStatus(PSM_UNLOADING_SNAPSHOTS);
55502|      // store the clusters of where the files are so
| we can do direct disk io
55503|      StoreClustersOfFiles(Volume);
55504|
55505|      // let the system know that we should not do
| everything
55506|      Debug(DEBUG_DEVCON,("UnloadSnapShotsForVolume:
| directio=%d, Setting to true\n",DevExt->DoDirectIO));
55507|      DevExt->InLoadUnload = TRUE;
55508|      DevExt->DoDirectIO=TRUE;
55509|
55510|      // dismount all virtual volumes
55511|      GetSnapShotForRead();
55512|      __try {
55513|          Rebuild_DismountAllVolumes(Volume,TRUE);
55514|      } __finally {
55515|          ReleaseSnapShotForRead();
55516|          if(AbnormalTermination()) {
55517|              Rebuild_ReenableVolumeMounts(Volume);
55518|          }
55519|      }

```

```

55520|
55521|    // remove all the virtual volumes
55522|    RemoveVolumesFromSystem(Volume);
55523|
55524|    // remove snapshot droppings
55525|    //RemoveSnapShotDroppings(Volume);
55526|
55527|    // remove snapshots from linked list (if any on
    | it)
55528|    PLIST_ENTRY ListEntry =
    | DevExt->Cache.SnapShotHead.Flink;
55529|    while(ListEntry!=&DevExt->Cache.SnapShotHead) {
55530|        pInternalSnapShot
    | SnapShot=CONTAINING_RECORD(ListEntry,tInternalSnapShot,L
    | istEntry);
55531|        RemoveSnapShotFromList(SnapShot);
55532|        FreeSnapShot(&SnapShot);
55533|        ListEntry =
    | DevExt->Cache.SnapShotHead.Flink;
55534|    }
55535|
55536|    // make sure to NOT call SaveHeader or all the
    | snapshots will
55537|    // go away
55538|
55539|    // remove memory and close cache files (if any,
    | as
55540|    // the RemoveVolumesFromSystem above should
    | have
55541|    // cleared this out.
55542|    TearDownCacheForVolume(Volume);
55543|    DevExt->Cache.ReferenceCount = 0;
55544|    } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
55545|        Status = GetExceptionCode();
55546|        Debug(DEBUG_DICT,("Exception %08x in
    | pd::UnloadSnapShotsForVolume\n",Status));
55547|    }
55548|    DevExt->InLoadUnload = FALSE;
55549|    UpdateGlobalStatus(PSM_IDLE);
55550|    if(!OpenCloseOwned) {
55551|        ReleaseOpenCloseResource();
55552|    }
55553|
55554|    Debug(DEBUG_DICT,("pd::UnloadSnapShotsForVolume
    | returning %08x\n",Status));
55555| #endif
55556|    return Status;
55557| }
55558|

```

```

55559| NTSTATUS
    | PersistentDictionary::MiniUnloadSnapShotsForVolume(
    | PDEVICE_OBJECT Volume, ULONG OpenCloseOwned )
55560| {
55561|     NTSTATUS Status=STATUS_SUCCESS;
55562|     #if 1
55563|         PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
55564|
55565|         // this is in the context of the calling thread.
    | IE autochk or mount.exe, etc..
55566|         // do not assume GlobalSystemProcessId ==
    | PsGetCurrentProcess()
55567|
55568|         | Debug(DEBUG_DICT,("pd::MiniUnloadSnapShotsForVolume:
    | Volume=%08x, '%S'\n", Volume, DevExt->Name));
55569|
55570|         DevExt->Dismounting = TRUE;
55571|
55572|         if(!OpenCloseOwned) {
55573|             // cancel all pending creates
55574|             CancelCreationOfSnapShots();
55575|
55576|             if (
    | AcquireOpenCloseResourceOnly(NULL)!=STATUS_WAIT_0 ) {
55577|                 DevExt->Dismounting = FALSE;
55578|                 return PSM_CANCELED_BY_USER;
55579|             }
55580|         }
55581|
55582|         __try {
55583|             UpdateGlobalStatus(PSM_UNMAPPING_SNAPSHOTS);
55584|
55585|             // store the clusters of where the files are so
    | we can do direct disk io
55586|             StoreClustersOfFiles(Volume);
55587|
55588|             // let the system know that we should not do
    | everything
55589|
    | Debug(DEBUG_DEVCON,("MiniUnloadSnapShotsForVolume:
    | directio=%d, Setting to true\n", DevExt->DoDirectIO));
55590|             DevExt->InLoadUnload = TRUE;
55591|
55592|             // dismount all virtual volumes
55593|             GetSnapShotForRead();
55594|             __try {
55595|                 Rebuild_DismountAllVolumes(Volume,TRUE);
55596|             } __finally {

```

```

55597|         ReleaseSnapShotForRead();
55598|         if(AbnormalTermination()) {
55599|             Rebuild_ReenableVolumeMounts(Volume);
55600|         }
55601|     }
55602|     DeleteVirtualVolumesFromSystem(Volume);
55603|
55604|     DevExt->DoDirectIO=TRUE;
55605|
55606|     | DeleteFileObjectToSkip(DevExt->Cache.HeaderFile.FileObjec
55607|     | ct);
55608|     | DeleteFileObjectToSkip(DevExt->Cache.IndexFile.FileObjec
55609|     | t);
55610|     | DeleteFileObjectToSkip(DevExt->Cache.CacheFile.FileObjec
55611|     | t);
55612|     CloseFilesForVolume(Volume);
55613| } __except(
55614|     | ExceptionFilter(GetExceptionInformation()) ) {
55615|     Status = GetExceptionCode();
55616|     Debug(DEBUG_DICT,("Exception %08x in
55617|     | pd::MiniUnloadSnapShotsForVolume\n",Status));
55618| }
55619| DevExt->Dismounting = FALSE;
55620| DevExt->InLoadUnload = FALSE;
55621| UpdateGlobalStatus(PSM_IDLE);
55622| if(!OpenCloseOwned) {
55623|     ReleaseOpenCloseResource();
55624| }
55625|
55626| Debug(DEBUG_DICT,("pd::MiniUnloadSnapShotsForVolume
55627| | returning %08x\n",Status));
55628| #endif
55629| return Status;
55630| }
55631|
55632| //-----
55633| | -----
55634| // class VirtualWritesNodeFinder allows us to do a
55635| | recursive traversal of
55636| // a virtual writes tree to find a batch of keys to be
55637| | deleted.
55638| // Why a class for this? Because it greatly reduces
55639| | stack space when we do
55640| // recursion, since we can pass a lot fewer parameters
55641| | in the recursive part.

```

```

55633| //-----
55634| | -----
55634| class VirtualWritesNodeFinder
55635| {
55636| public:
55637|     VirtualWritesNodeFinder ( tTreeLeaf *_NilNode,
55638|         | ULONG _SequenceNumber, keyType *_DeleteTable, ULONG
55639|         | _TableSize ):
55638|         NilNode(_NilNode),
55639|         SequenceNumber(_SequenceNumber),
55640|         DeleteTable(_DeleteTable),
55641|         NumInTable(0),
55642|         TableSize(_TableSize)
55643|     {}
55644|
55645| #ifdef DEBUG
55646|     // The following constructor is used only for doing
55647|     | debug dump
55647|     VirtualWritesNodeFinder ( tTreeLeaf *_NilNode ):
55648|         NilNode(_NilNode),
55649|         SequenceNumber(0),
55650|         DeleteTable(NULL),
55651|         NumInTable(0),
55652|         TableSize(0)
55653|     {}
55654| #endif /*DEBUG*/
55655|
55656|     ULONG FindBatch ( tTreeLeaf *HeadNode ); //
55657|     | returns the number of nodes found (will be
55658|     | 0..TableSize)
55659|
55658| #ifdef DEBUG
55659|     void DebugDump ( tTreeLeaf *Node, ULONG Depth );
55660| #endif /*DEBUG*/
55661|
55662| protected:
55663|     bool FindVirtualWriteNodesToDelete ( tTreeLeaf
55664|         | *NodeInTree, ULONG RecursionDepth );
55665|
55665| private:
55666|     tTreeLeaf *_NilNode;
55667|     ULONG     SequenceNumber;
55668|     keyType   *DeleteTable;
55669|     ULONG     NumInTable;
55670|     ULONG     TableSize;
55671| };
55672|
55673|
55674| //-----
55675| | -----

```

```

55675| #ifdef DEBUG
55676| void VirtualWritesNodeFinder::DebugDump ( tTreeLeaf
    | *node, ULONG depth )
55677| {
55678|     if ( depth < 32 ) {
55679|         if ( node != NilNode ) {
55680|             DebugDump (node->Left, 1+depth);
55681|             Debug(DEBUG_DICT,("VWT Dump: Depth=%2x
    | Key=%016I64x Pos=%08x\n",depth,node->Key,node->Pos));
55682|             DebugDump (node->Right, 1+depth);
55683|         }
55684|     } else {
55685|         ASSERT(FALSE);
55686|     }
55687| }
55688| #endif /*DEBUG*/
55689| //-----
    | -----
55690|
55691| ULONG VirtualWritesNodeFinder::FindBatch ( tTreeLeaf
    | *HeadNode )
55692| {
55693|     NumInTable = 0;
55694|
55695|     if ( DeleteTable!=NULL && TableSize>0 ) {
55696|         if ( HeadNode != NilNode ) {
55697|             FindVirtualWriteNodesToDelete ( HeadNode, 0
    | );
55698|         }
55699|     } else {
55700|         ASSERT(DeleteTable!=NULL);
55701|         ASSERT(TableSize>0);
55702|     }
55703|
55704|     return NumInTable;
55705| }
55706|
55707| //-----
    | -----
55708|
55709| bool
    | VirtualWritesNodeFinder::FindVirtualWriteNodesToDelete
    | (
55710|     tTreeLeaf *NodeInTree,
55711|     ULONG RecursionDepth )
55712| {
55713|     bool BailOut = false;
55714|     ASSERT ( NodeInTree != NilNode );
55715|
55716|     if ( RecursionDepth < 32 ) {

```

```

55717|    LARGE_INTEGER Key;
55718|    Key.QuadPart = NodeInTree->Key;
55719|    if ( Key.SnapShotPart == SequenceNumber ) {
55720|        ASSERT(NumInTable < TableSize);
55721|        DeleteTable[NumInTable++] = Key.QuadPart;
55722|        if ( NumInTable >= TableSize ) {
55723|            BailOut = true;
55724|        }
55725|    }
55726|
55727|    if ( !BailOut ) {
55728|        if ( NodeInTree->Left != NilNode ) {
55729|            BailOut = FindVirtualWriteNodesToDelete
| (NodeInTree->Left, 1+RecursionDepth);
55730|        }
55731|
55732|        if ( !BailOut ) {
55733|            if ( NodeInTree->Right != NilNode ) {
55734|                BailOut =
| FindVirtualWriteNodesToDelete (NodeInTree->Right,
| 1+RecursionDepth);
55735|            }
55736|        }
55737|    }
55738|    } else {
55739|        ASSERT(FALSE);    // we have gone deeper in
| recursion than makes any sense for our rbtrees!
55740|        BailOut = true;
55741|    }
55742|
55743|    return BailOut;
55744| }
55745|
55746| //-----
| -----
55747|
55748| NTSTATUS
| PersistentDictionary::RemoveVirtualWritesFromTree (
| ULONG ShouldEraseIndexSectors )
55749| {
55750|     NTSTATUS status = STATUS_SUCCESS;
55751|     ULONG SequenceNumber    = GetSequenceNumber();
55752|
55753|     Debug(DEBUG_DICT,("pd::RemoveVirtualWritesFromTree:
| Sequence=%08x, DevExt=%08x, Tree=%08x,
| TreeResource=%08x\n",SequenceNumber,DevExt,&Shared->Virt
| ualWritesTree,&Shared->TreeResource));
55754| #ifdef DEBUG
55755|     //Debug(DEBUG_DICT,("Virtual Writes Tree before
| removing nodes

```



```

| -----\n"));
55756| //DumpVirtualWriteTree();
55757| #endif /*DEBUG*/
55758|
55759| __try {
55760|     const ULONG BATCH_SIZE = 128; // how many
| nodes to process in one gulp
55761|     keyType *DeleteTable = (keyType
| *)MemAllocatePoolWithTag(PagedPool,sizeof(keyType)*BATCH
| _SIZE,TEMPTAG);
55762|     if ( DeleteTable ) {
55763|         ULONG NumInTable = 0;
55764|         do {
55765|             MyAcquireResourceSharedLite
| (&(Shared->TreeResource), TRUE);
55766|             __try {
55767|                 VirtualWritesNodeFinder NodeFinder
| (
55768|
| Shared->VirtualWritesTree.TailLeaf,
55769|                 SequenceNumber,
55770|                 DeleteTable,
55771|                 BATCH_SIZE );
55772|
55773|                 NumInTable = NodeFinder.FindBatch
| (Shared->VirtualWritesTree.HeadLeaf);
55774|             } __finally {
55775|                 MyReleaseResourceForThreadLite
| (&(Shared->TreeResource));
55776|             }
55777|
55778|             MyAcquireResourceExclusiveLite
| (&(Shared->TreeResource), TRUE);
55779|             __try {
55780|                 for ( ULONG i=0; i<NumInTable; ++i
| ) {
55781|                     tTreeLeaf *KillNode =
| rbtree_Delete (&(Shared->VirtualWritesTree),
| DeleteTable[i]);
55782|                     if ( KillNode ) {
55783|
| Debug(DEBUG_DICT,("RemoveVirtualWritesFromTree:
| Deleting key=%016l64x\n",DeleteTable[i]));
55784|                     if (
| ShouldEraseIndexSectors ) {
55785|                         NTSTATUS eraseStatus =
| EraseIndexInfo (DevExt, KillNode->Pos);
55786|
| ASSERT(NT_SUCCESS(eraseStatus));
55787|                     }

```

```

55788|             FreeNodeAndBits(KillNode);
55789|         } else {
55790|
55791|             | Debug(DEBUG_DICT,("RemoveVirtualWritesFromTree: Key
55792|             | %016l64x is missing from tree!\n",DeleteTable[i]));
55793|             ASSERT(FALSE);
55794|         }
55795|         } __finally {
55796|             MyReleaseResourceForThreadLite
55797|             | (&(Shared->TreeResource));
55798|         }
55799|         } while ( NumInTable > 0 );
55800|         MemFreePool (DeleteTable);
55801|         DeleteTable = NULL;
55802|     } else {
55803|         status = STATUS_INSUFFICIENT_RESOURCES;
55804|         ASSERT(FALSE);
55805|     }
55806| } __except(
55807|     | ExceptionFilter(GetExceptionInformation()) ) {
55808|     status = GetExceptionCode();
55809|     Debug(DEBUG_DICT,("!!!
55810|     | RemoveVirtualWritesFromTree: exception
55811|     | %08x\n",status));
55812| }
55813|
55814| #ifdef DEBUG
55815| //Debug(DEBUG_DICT,("Virtual Writes Tree after
55816| //removing nodes
55817| //-----\n"));
55818| //DumpVirtualWriteTree();
55819| #endif /*DEBUG*/
55820|
55821| Debug(DEBUG_DICT,("pd::RemoveVirtualWritesFromTree
55822| | returning %08x\n",status));
55823| return status;
55824| }
55825|
55826| //-----
55827| | -----
55828|
55829| #ifdef DEBUG
55830| void PersistentDictionary::DumpVirtualWriteTree()
55831| {
55832|     __try {
55833|         MyAcquireResourceSharedLite
55834|         | (&(Shared->TreeResource), TRUE);
55835|     } __try {
55836|         VirtualWritesNodeFinder dumper

```

```

    | (Shared->VirtualWritesTree.TailLeaf);
55827|         dumper.DebugDump
    | (Shared->VirtualWritesTree.HeadLeaf, 0);
55828|     } __finally {
55829|         MyReleaseResourceForThreadLite
    | (&(Shared->TreeResource));
55830|     }
55831| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
55832|     NTSTATUS status = GetExceptionCode();
55833|     Debug(DEBUG_DICT,("!!!
    | pd::DumpVirtualWriteTree: exception %08x\n",status));
55834| }
55835| }
55836| #endif /*DEBUG*/
55837|
55838| //-----
    | -----
55839|
55840| bool GenericTreeSearcher::visitEveryNode()
55841| {
55842|     bool searchCompleted = true;
55843|
55844|     if ( tree.HeadLeaf != tree.TailLeaf ) {
55845|         searchCompleted = recursiveTraversal
    | (tree.HeadLeaf, 0);
55846|     }
55847|
55848|     return searchCompleted;
55849| }
55850|
55851| //-----
    | -----
55852|
55853| bool GenericTreeSearcher::recursiveTraversal (
    | tTreeLeaf *node, int depth )
55854| {
55855|     ASSERT ( depth < 32 );
55856|
55857|     bool continueSearch = true;
55858|
55859|     if ( node->Left != tree.TailLeaf ) {
55860|         continueSearch = recursiveTraversal (
    | node->Left, 1+depth );
55861|     }
55862|
55863|     if ( continueSearch ) {
55864|         continueSearch = nodeCallback (node, depth);
55865|
55866|         if ( continueSearch ) {

```

```

55867|         if ( node->Right != tree.TailLeaf ) {
55868|             continueSearch = recursiveTraversal (
55869|                 | node->Right, 1+depth );
55870|         }
55871|     }
55872|
55873|     return continueSearch;
55874| }
55875|
55876| //-----
55877| | -----
55878| class TreeSearcher_MakeVirtualWritesPersistent: public
55879|     | GenericTreeSearcher
55880| {
55881| public:
55882|     TreeSearcher_MakeVirtualWritesPersistent (
55883|         tTree          &_tree,
55884|         PersistentDictionary &_dictionary,
55885|         PFILTERED_EXTENSION _devExt,
55886|         ULONG             _sequenceNumber ):
55887|             GenericTreeSearcher(_tree),
55888|             dictionary(_dictionary),
55889|             devExt(_devExt),
55890|             sequenceNumber(_sequenceNumber),
55891|             status(STATUS_SUCCESS)
55892|     {}
55893|
55894|     virtual bool nodeCallback ( tTreeLeaf *node, int
55895|         | depth );
55896|
55897|     NTSTATUS getStatus() const { return status; }
55898|
55899| private:
55900|     PersistentDictionary &dictionary;
55901|     PFILTERED_EXTENSION devExt;
55902|     ULONG             sequenceNumber;
55903|     NTSTATUS          status;
55904| };
55905|
55906| bool
55907|     | TreeSearcher_MakeVirtualWritesPersistent::nodeCallback
55908|     | (
55909|         tTreeLeaf *node,
55910|         int      depth )
55911| {
55912|     LARGE_INTEGER key;
55913|     key.QuadPart = node->Key;

```

```

55911|   if ( key.SnapShotPart == sequenceNumber ) {
55912|       status = dictionary.SaveIndexInfo (
55913|           devExt,
55914|           key.GranulePart,
55915|           sequenceNumber,
55916|           node->Pos,
55917|           CHECKSUM_IGNORE_DWORD,
55918|           PSM_INDEX_FLAG_VIRTUAL_WRITE );
55919|   }
55920|   return NT_SUCCESS(status) ? true : false;
55921| }
55922|
55923|
55924| //-----
55925| | -----
55926| NTSTATUS
55927| | PersistentDictionary::MakeVirtualWritesPersistent()
55928| {
55929|     NTSTATUS status = STATUS_SUCCESS;
55930|     __try {
55931|         Profile("pd::MakeVirtualWritesPersistent");
55932|         MyAcquireResourceSharedLite
55933|         | (&(Shared->TreeResource), TRUE);
55934|         __try {
55935|             TreeSearcher_MakeVirtualWritesPersistent
55936|             | saver (
55937|                 Shared->VirtualWritesTree,
55938|                 *this,
55939|                 DevExt,
55940|                 GetSequenceNumber() );
55941|             saver.visitEveryNode();
55942|             status = saver.getStatus();
55943|         } __finally {
55944|             MyReleaseResourceForThreadLite
55945|             | (&(Shared->TreeResource));
55946|         }
55947|     } __except(
55948|         | ExceptionFilter(GetExceptionInformation()) ) {
55949|         status = GetExceptionCode();
55950|         Debug(DEBUG_DICT,("!!!
55951|         | pd::MakeVirtualWritesPersistent: exception
55952|         | %08x\n",status));
55953|     }
55954| }
55955| return status;
55956| }
55957| //-----

```

```

| -----
55953|
55954| ULONG PersistentDictionary::QueryResetPsm()
55955| {
55956|     ULONG flag = FALSE;
55957|     if ( !ResetPsm_Disabled ) {
55958|         flag = ResetPsm;
55959|     }
55960|     Debug(DEBUG_DICT,("pd::QueryResetPsm returning
    | %s\n", (flag?"TRUE":"FALSE")));
55961|     return flag;
55962| }
55963|
55964| //-----
    | -----
55965|
55966| ULONG PersistentDictionary::QueryNoPsm()
55967| {
55968|     ULONG flag = FALSE;
55969|     if ( !NoPsm_Disabled ) {
55970|         flag = NoPsm;
55971|     }
55972|     Debug(DEBUG_DICT,("pd::QueryNoPsm returning %s\n",
    | (flag?"TRUE":"FALSE")));
55973|     return flag;
55974| }
55975|
55976| //-----
    | -----
55977|
55978| void PersistentDictionary::DisableResetPsm()
55979| {
55980|     ResetPsm_Disabled = TRUE;
55981|     Debug(DEBUG_DICT,("pd::DisableResetPsm()\n"));
55982| }
55983|
55984| //-----
    | -----
55985|
55986| void PersistentDictionary::DisableNoPsm()
55987| {
55988|     NoPsm_Disabled = TRUE;
55989|     Debug(DEBUG_DICT,("pd::DisableNoPsm()\n"));
55990| }
55991|
55992| //-----
    | -----
55993|
55994| PRTL_BITMAP PersistentDictionary::GetVolumeCachingMap(
    | ULONG Map )

```

```

55995| {
55996|     switch(Map) {
55997|         case 0 :return Shared->Map;
55998|         case 1: return Shared->MapInTransform;
55999|         default:
56000|             ASSERT(FALSE);
56001|     }
56002|     return NULL;
56003| }
56004|
56005| #if DO_ALL_BITMAPS
56006| void DumpCachingMap( PRTL_BITMAP *BitMap )
56007| {
56008|     if (*BitMap!=NULL) {
56009|         for (ULONG i=0;
56010|              | i*4<((*BitMap)->SizeOfBitMap+7)/8;i+=0x8 ) {
56011|             if ( (i==0) ||
56012|                  ( (*BitMap)->Buffer[i+0] !=
56013|                    | (*BitMap)->Buffer[i-8+0] ) ||
56014|                  ( (*BitMap)->Buffer[i+1] !=
56015|                    | (*BitMap)->Buffer[i-8+1] ) ||
56016|                  ( (*BitMap)->Buffer[i+2] !=
56017|                    | (*BitMap)->Buffer[i-8+2] ) ||
56018|                  ( (*BitMap)->Buffer[i+3] !=
56019|                    | (*BitMap)->Buffer[i-8+3] ) ||
56020|                  ( (*BitMap)->Buffer[i+4] !=
56021|                    | (*BitMap)->Buffer[i-8+4] ) ||
56022|                  ( (*BitMap)->Buffer[i+5] !=
56023|                    | (*BitMap)->Buffer[i-8+5] ) ||
56024|                  ( (*BitMap)->Buffer[i+6] !=
56025|                    | (*BitMap)->Buffer[i-8+6] ) ||
56026|                  ( (*BitMap)->Buffer[i+7] !=
56027|                    | (*BitMap)->Buffer[i-8+7] ) ) {
56028|             Debug(DEBUG_DEVSUP,(" Granule %08x:
56029|             | %08x %08x %08x %08x - %08x %08x %08x %08x\n"
56030|             | ,i*0x20
56031|             | ,(*BitMap)->Buffer[i+0x0]
56032|             | ,(i+1)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
56033|             | (*BitMap)->Buffer[i+0x1] : 0x0dd1b1d5
56034|             | ,(i+2)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
56035|             | (*BitMap)->Buffer[i+0x2] : 0x0dd1b1d5
56036|             | ,(i+3)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
56037|             | (*BitMap)->Buffer[i+0x3] : 0x0dd1b1d5
56038|             | ,(i+4)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
56039|             | (*BitMap)->Buffer[i+0x4] : 0x0dd1b1d5

```

```

56027|
| ,(i+5)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
| (*BitMap)->Buffer[i+0x5] : 0x0dd1b1d5
56028|
| ,(i+6)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
| (*BitMap)->Buffer[i+0x6] : 0x0dd1b1d5
56029|
| ,(i+7)*4<((*BitMap)->SizeOfBitMap+7)/8 ?
| (*BitMap)->Buffer[i+0x7] : 0x0dd1b1d5
56030|         ));
56031|     }
56032| }
56033| }
56034| }
56035| #endif
56036|
56037| void PersistentDictionary::SetVolumeCachingMap( ULONG
| Map , PRTL_BITMAP BitMap )
56038| {
56039|     switch(Map) {
56040|
56041|         case 0 :
56042|
56043| #if DO_ALL_BITMAPS
56044|         Debug(DEBUG_DEVSUP,(" Replacing active
| Granule Map at 0x%08x . . .\n", Shared->Map ));
56045|         DumpCachingMap( &Shared->Map );
56046| #endif
56047|         Shared->Map = BitMap;
56048| #if DO_ALL_BITMAPS
56049|         Debug(DEBUG_DEVSUP,(" . . . with this fresh
| Granule Map at 0x%08x . . .\n", Shared->Map ));
56050|         DumpCachingMap( &Shared->Map );
56051| #endif
56052|         break;
56053|
56054|         case 1:
56055|
56056| #if DO_ALL_BITMAPS
56057|         Debug(DEBUG_DEVSUP,(" Replacing Granule Map
| in transform at 0x%08x . . .\n", Shared->MapInTransform
| ));
56058|         DumpCachingMap( &Shared->MapInTransform );
56059| #endif
56060|         Shared->MapInTransform = BitMap;
56061| #if DO_ALL_BITMAPS
56062|         Debug(DEBUG_DEVSUP,(" . . . with this fresh
| Granule Map in transform at 0x%08x . . .\n",
| Shared->MapInTransform ));
56063|         DumpCachingMap( &Shared->MapInTransform );

```



```

56064| #endif
56065|         break;
56066|
56067|         default:
56068|             ASSERT(FALSE);
56069|     }
56070|     return;
56071| }
56072|
56073| ULONG PersistentDictionary::GetVolumeClusterSize(void)
56074| {
56075|     return Shared->ClusterSize;
56076| }
56077|
56078|
56079| /*--- end of file perdict.cpp ---*/
56080|
56081|
56082|
56083| File Listing: perdict_rebuild.cpp
56084|
56085| #include "precomp.h"
56086|
56087| #define ENABLE_INDEX_LOAD_WHEEL 1
56088| #define INVALID_HANDLE_VALUE ((HANDLE)(-1))
56089| #define INVALID_BIT_INDEX ((ULONG)(-1))
56090|
56091| //-----
56092| | -----
56093| DECLARE_STOPWATCH(Part2);
56094| DECLARE_STOPWATCH(Part2_AddingStage1);
56095| DECLARE_STOPWATCH(Part2_AddDrive);
56096| DECLARE_STOPWATCH(Part2_Credit);
56097| DECLARE_STOPWATCH(Part2_AddingStage2);
56098| DECLARE_STOPWATCH(Part2_DismountAll);
56099| DECLARE_STOPWATCH(Part2_MountAll);
56100| DECLARE_STOPWATCH(Part2_RecreateJunctions);
56101| DECLARE_STOPWATCH(Rebuild_Total);
56102| DECLARE_STOPWATCH(Rebuild_ReadFromFile);
56103| DECLARE_STOPWATCH(Rebuild_AllocNode);
56104| DECLARE_STOPWATCH(Rebuild_FindDict);
56105| DECLARE_STOPWATCH(Rebuild_Adding);
56106| DECLARE_STOPWATCH(Rebuild_SettingBits);
56107| DECLARE_STOPWATCH(Rebuild_AcquireExclusive);
56108| DECLARE_STOPWATCH(Rebuild_InsertToTree);
56109| DECLARE_STOPWATCH(Rebuild_FreeNode);
56110| DECLARE_STOPWATCH(Rebuild_FreeDeadNodes);
56111| DECLARE_STOPWATCH(Rebuild_RevertCheck);
56112|

```

```

56113| #if DEBUG_VALIDATE_DIFF_GRANULES
56114|   DECLARE_STOPWATCH(Rebuild_ValidateDiffGranules);
56115| #endif /*DEBUG_VALIDATE_DIFF_GRANULES*/
56116|
56117| //-----
| -----
56118|
56119| STATIC void LogCorruptIndexSector (
56120|   ULONG   sectorNumber,
56121|   NTSTATUS status )
56122| {
56123|   ULONG dumpData[] = { status, sectorNumber};
56124|   WCHAR sectorNumberString[32];
56125|   WCHAR *stringArray[] = { sectorNumberString};
56126|   _Itow ( sectorNumber, sectorNumberString, 10,
| sizeof(sectorNumberString)-1 );
56127|   LogError (
56128|     (PDEVICE_OBJECT)PSManDriverObject,
56129|     NULL,
56130|     PSM_CORRUPT_INDEX,
56131|     status,
56132|     dumpData,
56133|     sizeof(dumpData),
56134|     stringArray,
56135|     sizeof(stringArray) /
| sizeof(stringArray[0]));
56136|
56137|   Debug(DEBUG_DICT,("Recovery detected corrupt index
| sector %!64u\n",(ULONGLONG)sectorNumber));
56138| }
56139|
56140| //-----
| -----
56141|
56142| STATIC void LogTreeInsertionError (
56143|   pTreeLeaf node,
56144|   int insertResult )
56145| {
56146|   ULONG dumpData[] = { node->Pos};
56147|   WCHAR insertResultString[32];
56148|   WCHAR keyLowString[32];
56149|   WCHAR keyHighString[32];
56150|   WCHAR *stringArray[] = { insertResultString,
| keyLowString, keyHighString};
56151|   _Itow ( insertResult, insertResultString, 10,
| sizeof(insertResultString)-1 );
56152|   _Itow ( (ULONG)(node->Key), keyLowString, 16,
| sizeof(keyLowString)-1 );
56153|   _Itow ( (ULONG)(node->Key >> 32), keyHighString,
| 16, sizeof(keyHighString)-1 );

```

```

56154|   LogError (
56155|       (PDEVICE_OBJECT)PSManDriverObject,
56156|       NULL,
56157|       PSM_TREE_INSERT_ERROR,
56158|       0,
56159|       dumpData,
56160|       sizeof(dumpData),
56161|       stringArray,
56162|       sizeof(stringArray) /
        | sizeof(stringArray[0]));
56163|
56164|   Debug(DEBUG_DICT,("Recovery could not insert
        | key=0x%016l64x pos=%u into rbtree (error=%08x).\n",
56165|       node->Key, node->Pos,
        | insertResult));
56166| }
56167|
56168| //-----
        | -----
56169|
56170| STATIC void LogSnapshotEntryNotFound (
56171|     pkSnapShotMaster  m,
56172|     PDEVICE_OBJECT    VolObj,
56173|     ULONG              sectorNumber )
56174| {
56175|     ULONG dumpData[] = { (ULONG)m, (ULONG)VolObj};
56176|     LogError (
56177|         (PDEVICE_OBJECT)PSManDriverObject,
56178|         NULL,
56179|         PSM_SNAPSHOT_ENTRY_NOT_FOUND,
56180|         0,
56181|         dumpData,
56182|         sizeof(dumpData),
56183|         NULL,
56184|         0 );
56185|
56186|     Debug(DEBUG_DICT,("Error getting snapshot entry for
        | master %08x, volume %08x\n",m,VolObj));
56187| }
56188|
56189| //-----
        | -----
56190| // Given a snapshot set (master) and volume object,
        | will return back a snapshot (entry) for that volume.
56191| // One will be created if it does not exist
56192|
56193| struct skSnapShotEntry
        | *PersistentDictionary::GetSnapShotForMaster (
56194|     skSnapShotMaster  *MasterSnapShot,
56195|     PDEVICE_OBJECT    VolumeObject,

```

```

56196|  pInternalSnapShot  SnapShot)
56197| {
56198|  NTSTATUS Status=STATUS_SUCCESS;
56199|  pkSnapShotEntry p=NULL;
56200|  PFILTERED_EXTENSION VolExt=NULL;
56201|
56202|  // if volume object is known, scan to see if we
    | already have a snapshot for it
56203|  if ( VolumeObject ) {
56204|      VolExt = GetFilteredExtension (VolumeObject);
56205|
56206|      GetSnapShotForRead();
56207|      __try {
56208|
    | p=GetTopSnapShotForMaster(&MasterSnapShot->SnapShots);
56209|      while ( p ) {
56210|          if ( p->DeviceObject==VolumeObject ) {
56211|              DoneWithSnapShot(p);
56212|              break;
56213|          }
56214|
    | p=GetNextSnapShotForMaster(&MasterSnapShot->SnapShots,p)
    | ;
56215|      }
56216|      } __finally {
56217|          ReleaseSnapShotForRead();
56218|      }
56219|  }
56220|
56221|  // no snapshot for this volume, so lets create one
56222|  if ( !p ) {
56223|      #if DO_ALL_SEARCH
56224|          Debug(DEBUG_DICT,(
56225|              "pd::GetSnapShotForMaster:
    | Creating new snapshot for VolumeObject=%08x,
    | InternalSnapShot=%08x\n",
56226|              VolumeObject,
56227|              SnapShot));
56228|      #endif /*DO_ALL_SEARCH*/
56229|
56230|      // find system process
56231|      pOT_USER User =
    | FindPSMUser(GlobalSystemProcessId,(_ETHREAD*)-2);
56232|      ASSERT(User);
56233|
56234|      // need to make one
56235|      p = (tkSnapShotEntry *) MemAllocatePoolWithTag(
    | NonPagedPool,
    | sizeof(tkSnapShotEntry),PSM_SNAPSHOT_ENTRY);
56236|      if ( p ) {

```

```

56237|         RtlZeroMemory ( p, sizeof(tkSnapshotEntry)
| );
56238|         p->DeviceObject = VolumeObject;
56239|         p->MasterSnapShot = MasterSnapShot;
56240|         p->Deleted = FALSE;
56241|         p->Count=0;
56242|         p->PSMSectors = NULL;
56243|
56244|         // Must be persistent snapshot. Otherwise,
| why is it in rebuild?
56245|         | ASSERT(PSM_SS_IsPersistent(SnapShot->Permanent.SnapShotF
| lags));
56246|         ASSERT(!(SnapShot->Permanent.SnapShotFlags
| & PSM_SS_BIT_SAVE_TEMP_ON_EXIT));
56247|         Status = Dictionary::Open
| (DICT_FLAG_PERSISTENT, p->Dictionary);
56248|         if ( NT_SUCCESS(Status) ) {
56249|             PDEVICE_OBJECT Virtual = 0;
56250|
56251|             ULONG CurrentSequenceNumber =
| SnapShot->Permanent.SequenceNumber;
56252|             #if DO_ALL_SEARCH
56253|             | Debug(DEBUG_DICT,("pd::GetSnapShotForMaster:
| snapshot=%08x,
| sequence=%08x\n",p,CurrentSequenceNumber));
56254|             #endif /*DO_ALL_SEARCH*/
56255|
56256|             | ((pPersistentDictionary)p->Dictionary)->SetVolumeInterna
| l(
56257|                 VolumeObject,
56258|                 SnapShot,
56259|                 CurrentSequenceNumber );
56260|
56261|             InterlockedIncrement((PLONG)
| &MasterSnapShot->Count);
56262|
56263|             if ( VolumeObject ) {
56264|                 VolExt->SignalWrite = 0;
56265|                 InterlockedIncrement((PLONG)
| &VolExt->PSMed);
56266|                 InterlockedIncrement((PLONG)
| &VolExt->OpenCount);
56267|
56268|                 GetSnapShotForRead();
56269|                 __try {
56270|                     pkSnapShotEntry
| q=GetTopSnapShot(&VolExt->SnapShots);

```

```

56271|         while ( q ) {
56272|             if (
56273|                 | ((pPersistentDictionary)(q->Dictionary))->GetSequenceNum
56274|                 | ber() >
56275|                 | ((pPersistentDictionary)(p->Dictionary))->GetSequenceNum
56276|                 | ber() ) {
56277|
56278| #define InsertPrecedingInList(Entry,New) {\
56279|     (New)->Flink = (Entry);\
56280|     (New)->Blink = (Entry)->Blink;\
56281|     (Entry)->Blink->Flink = (New);\
56282|     (Entry)->Blink = (New);\
56283| }
56284|
56285| | InsertPrecedingInList(&q->DevExt,&p->DevExt);
56286| DoneWithSnapShot(q);
56287| break;
56288| }
56289|
56290| | q=GetNextSnapShot(&VolExt->SnapShots,q);
56291| }
56292|
56293| if ( q==NULL ) {
56294|     //we're the first but
56295|     | insert-pre works cos the list is circular!!
56296|
56297| | InsertPrecedingInList(&VolExt->SnapShots,&p->DevExt);
56298| }
56299|
56300| } __finally {
56301|     ReleaseSnapShotForRead();
56302| }
56303|
56304| Insert ...here...
56305| //
56306| | InsertHeadList(&VolExt->SnapShots,&p->DevExt);
56307|
56308| }
56309|
56310| | InsertHeadList(&MasterSnapShot->SnapShots,&p->Master);
56311|
56312| | InsertHeadList(&User->SnapShots,&p->User);
56313| InterlockedIncrement((PLONG)
56314| | &User->NumOpenSnapShots);
56315| } else {

```

```

56309|         Debug(DEBUG_DICT,("Error %08x creating
| dictionary\n",Status));
56310|     }
56311| } else {
56312|     Debug(DEBUG_DICT,("Out of memory getting
| snapshot for master\n"));
56313| }
56314| }
56315|
56316| return p;
56317| }
56318|
56319| //-----
| -----
56320|
56321| pkSnapShotMaster
| PersistentDictionary::FindMasterSnapShotOnOtherVolume (
56322| pInternalSnapShot SnapShot )
56323| {
56324|     pkSnapShotMaster master = 0;
56325|     ULONG NumDictionariesInList = 0;
56326|
56327|     #if DO_ALL_SEARCH
56328|
| Debug(DEBUG_DICT,("pd::FindMasterSnapShotOnOtherVolume:
| SnapShot=%08x,
| Sequence=%08x\n",SnapShot,SnapShot->Permanent.SequenceNu
| mber));
56329|     #endif /*DO_ALL_SEARCH*/
56330|
56331|     if ( SnapShot->Permanent.SequenceNumber > 0 ) {
56332|
| //Debug(DEBUG_DICT,("pd::FindMasterSnapShotOnOtherVolume
| : SnapShot=%08x, Sequence=%08x, Time=%016l64x -
| looking through all
| dictionaries\n",SnapShot,SnapShot->SequenceNumber,SnapSh
| ot->SnapShotTime.QuadPart));
56333|     ASSERT (
| SnapShot->Permanent.SnapShotTime.QuadPart > 0 );
56334|
56335|     // Search through all dictionaries to find a
| master with the same sequence number and time.
56336|
56337|     pPersistentDictionary dict = ListHead;
56338|     while ( dict != NULL ) {
56339|         if ( dict->SnapShot &&
| dict->SnapShot!=SnapShot &&
| dict->SnapShot->SnapShotMaster ) {
56340|             ULONG SameSequence =
| (dict->SnapShot->Permanent.SequenceNumber ==

```

```

    | SnapShot->Permanent.SequenceNumber);
56341|         ULONG SameTime    =
    | (dict->SnapShot->Permanent.SnapShotTime.QuadPart ==
    | SnapShot->Permanent.SnapShotTime.QuadPart);
56342|         if ( /*SameSequence &&*/ SameTime ) {
56343|
    | //Debug(DEBUG_DICT,("pd::FindMasterSnapShotOnOtherVolume
    | : Matching dictionary: SnapShot=%08x, Sequence=%08x,
    | Time=%016l64x,
    | ListIndex=%08x\n",dict->SnapShot,dict->SnapShot->Sequenc
    | eNumber,dict->SnapShot->SnapShotTime.QuadPart,NumDiction
    | ariesInList));
56344|         master =
    | dict->SnapShot->SnapShotMaster;
56345|         ASSERT(SameSequence); //not
    | horrible, but I want to know about it when it happens
56346|         if ( !SameSequence ) {
56347|
    | SnapShot->Permanent.SequenceNumber =
    | dict->SnapShot->Permanent.SequenceNumber; //Experiment
56348|         }
56349|         break;
56350|     }
56351| }
56352|
56353|     dict = dict->Next;
56354|     ++NumDictionariesInList;
56355| }
56356|
56357| #ifdef DEBUG
56358|     #if DO_ALL_SEARCH
56359|         while ( dict != NULL ) {
56360|             dict = dict->Next;
56361|             ++NumDictionariesInList;
56362|         }
56363|
    | Debug(DEBUG_DICT,("pd::FindMasterSnapShotOnOtherVolume:
    | NumDictionaries=%08x\n",NumDictionariesInList));
56364|     #endif
56365| #endif /*DEBUG*/
56366| }
56367|
56368| return master;
56369| }
56370|
56371| //-----
    | -----
56372| // Given an index number, will return back a snapshot
    | set (master) associated with that index.
56373| // If one doesnt exist it will create it.

```



```

56374|
56375| struct skSnapShotMaster
    | *PersistentDictionary::GetAMasterSnapShot (
56376|     struct _OT_USER_ *User,
56377|     pInternalSnapShot SnapShot )
56378| {
56379|     //Debug(DEBUG_DICT,("pd::GetAMasterSnapShot called:
    | SnapShot=%08x\n",SnapShot));
56380|
56381|     // change to allow the extra one used as a dummy
    | with all volumes on which
56382|     // to hang deceased snapshots
56383|     if ( !SnapShot ) {
56384|         return NULL;
56385|     }
56386|
56387|     //Debug(DEBUG_DICT,("pd::GetAMasterSnapShot:
    | SnapShot->SnapShotMaster =
    | %08x\n",SnapShot->SnapShotMaster));
56388|
56389|     if ( !SnapShot->SnapShotMaster ) {
56390|         SnapShot->SnapShotMaster =
    | FindMasterSnapShotOnOtherVolume (SnapShot);
56391|         if ( SnapShot->SnapShotMaster ) {
56392|             #if DO_ALL_SEARCH
56393|
    | Debug(DEBUG_DICT,("pd::GetAMasterSnapShot: Found other
    | volume's master %08x to link to internal
    | %08x\n",SnapShot->SnapShotMaster,SnapShot));
56394|             #endif /*DO_ALL_SEARCH*/
56395|         } else {
56396|             SnapShot->SnapShotMaster =
    | (tkSnapShotMaster *) MemAllocatePoolWithTag(
    | NonPagedPool,
    | sizeof(tkSnapShotMaster),PSM_MASTER_SNAPSHOT);
56397|
    | if ( SnapShot->SnapShotMaster ) {
56398|         RtlZeroMemory(
    | SnapShot->SnapShotMaster, sizeof(tkSnapShotMaster));
56400|
56401|         InitializeListHead(&SnapShot->SnapShotMaster->SnapShots)
    | ;
56402|
56403|         SnapShot->SnapShotMaster->ExclusiveProcess = 0;
56404|         SnapShot->SnapShotMaster->OutOfSeconds
    | = 0;
56405|         SnapShot->SnapShotMaster->Count
    | = 0;

```

```

56406|         SnapShot->SnapShotMaster->Status
      | = STATUS_SUCCESS;
56407|         SnapShot->SnapShotMaster->Persistent
      | = TRUE;    // Called only by rebuild - thus all
      | snapshots are persistent.
56408|
56409|         if (
      | SnapShot->Permanent.SequenceNumber==0 ) {
56410|         | SnapShot->SnapShotMaster->SnapShotTime.QuadPart = 0;
56411|         | SnapShot->SnapShotMaster->GroupNumber      = 0;
56412|         | SnapShot->SnapShotMaster->NumToKeep
      | = -1;
56413|         | SnapShot->SnapShotMaster->Priority
      | = 0;
56414|         | SnapShot->SnapShotMaster->SnapShotFlags     = 0;
56415|         | SnapShot->SnapShotMaster->DllPrivateUse     = 0;
56416|         | wcscpy(SnapShot->SnapShotMaster->UserSnapShotName,L "");
56417|         | SnapShot->SnapShotMaster->Instance
      | = MAX_NUMBER_OF_SNAPSHOTS;
56418|         } else {
56419|         | SnapShot->SnapShotMaster->DllPrivateUse     =
      | SnapShot->DllPrivateUse;
56420|
56421|         | SnapShot->SnapShotMaster->GroupNumber =
      | SnapShot->Permanent.GroupNumber;
56422|         | SnapShot->SnapShotMaster->Status
      | = SnapShot->Permanent.Status;
56423|         | SnapShot->SnapShotMaster->SnapShotTime =
      | SnapShot->Permanent.SnapShotTime;
56424|         | SnapShot->SnapShotMaster->NumToKeep
      | = SnapShot->Permanent.NumToKeep;
56425|         | SnapShot->SnapShotMaster->Priority
      | = SnapShot->Permanent.Priority;
56426|         | SnapShot->SnapShotMaster->SnapShotFlags =
      | SnapShot->Permanent.SnapShotFlags;
56427|         | SnapShot->SnapShotMaster->Instance
      | = SnapShot->Permanent.ExternalInstance;
56428|         | wcscpy(SnapShot->SnapShotMaster->UserSnapShotName, SnapSh
      | ot->Permanent.UserSnapShotName);
56429|

```

```

56430|         if (
| Snapshot->SnapshotMaster->Instance>=VDiskInstance ) {
56431|             VDiskInstance = 1 +
| Snapshot->SnapshotMaster->Instance;
56432|         }
56433|     }
56434|
56435|         InterlockedIncrement((PLONG)
| &GlobalData->NumActive);
56436|         InterlockedIncrement((PLONG)
| &User->Open);
56437|         #if DO_ALL_SEARCH
56438|         | Debug(DEBUG_DICT,("pd::GetAMasterSnapShot: Incremented
| NumActive to %08x
| open=%08x\n",GlobalData->NumActive,User->Open));
56439|         #endif /*DO_ALL_SEARCH*/
56440|     } else {
56441|         Debug(DEBUG_DICT,("Error out of memory
| for master snapshot\n"));
56442|     }
56443| }
56444| } else {
56445|     // already allocated
56446| }
56447|
56448| // VerifyHeaderIntegrity ( Header, "global header
| after GetAMasterSnapShot" );
56449| //Debug(DEBUG_DICT,("pd::GetAMasterSnapShot
| returning master %08x\n",Snapshot->SnapshotMaster));
56450| return Snapshot->SnapshotMaster;
56451| }
56452|
56453|
56454| //-----
| -----
56455| // Given an ordered pair (Lower,Upper) of snapshot
| numbers, traverses the whole Rbtree and
56456| // sets as handled (a 0 bit) in the appropriate volume
| bitmap each granule that was cached
56457| // after the lower snapshot and before the upper
| snapshot was taken.
56458| // (NB: Upper = 0 means we accept any Sequence no.
| above Lower)
56459|
56460| void
| PersistentDictionary::CreditInterveningGranulesInTree (
56461|     PRTL_BITMAP GranuleBitMap,
56462|     tTree *Tree,
56463|     ULONG LowerSS,

```

```

56464|    ULONG        UpperSS )
56465| {
56466|    __try {
56467|        ULARGE_INTEGER CurrentKey;
56468|        tTreeLeaf *CurrentNode =
56469|            | rbtree_SearchUpperBound(Tree, 0);
56470|        CurrentKey.QuadPart = CurrentNode->Key;
56471|
56472|        #define StepThroughTree CurrentNode =
56473|            | rbtree_GetNextInOrder(Tree, CurrentNode); if
56474|            | (CurrentNode) {CurrentKey.QuadPart = CurrentNode->Key;}
56475|        while ( CurrentNode!=NULL ) {
56476|            //advance to next granule with a snapshot
56477|            | at least high enough to be worth inspecting
56478|            while ( (CurrentNode!=NULL) &&
56479|                | (CurrentKey.SnapShotPart < LowerSS) ) {
56480|                #if DO_ALL_FREESPACE
56481|                | Debug(DEBUG_DICT,("pd::CreditInterveningGranulesInTree:
56482|                | scope: |%x - %x Skipping %8x|%08x - SS too low\n"
56483|                | , LowerSS
56484|                | , UpperSS
56485|                | ,
56486|                | CurrentKey.GranulePart
56487|                | ,
56488|                | CurrentKey.SnapShotPart
56489|                | ));
56490|                #endif /*DO_ALL_FREESPACE*/
56491|                StepThroughTree;
56492|            }
56493|            // if in the intervening range then credit
56494|            | it as already snapped
56495|            ULONG Granule = CurrentKey.GranulePart;
56496|            if ( (CurrentNode!=NULL) &&
56497|                | ((CurrentKey.SnapShotPart < UpperSS) || (UpperSS==0 ))
56498|                | ) {
56499|                // free bit
56500|                if(GranuleBitMap) {
56501|                    PsmBitPositionValidate
56502|                    | (GranuleBitMap, Granule);
56503|                    RtlClearBits ( GranuleBitMap,
56504|                    | Granule, 1 );
56505|                }
56506|                #if DO_ALL_FREESPACE
56507|                | Debug(DEBUG_DICT,("pd::CreditInterveningGranulesInTree:

```

```

    | scope: |%x - %x Crediting %8x|%08x\n"
56499|                                     , LowerSS
56500|                                     , UpperSS
56501|                                     ,
    | CurrentKey.GranulePart
56502|                                     ,
    | CurrentKey.SnapShotPart
56503|                                     ));
56504|         #endif /*DO_ALL_FREESPACE*/
56505|         StepThroughTree;
56506|     }
56507|
56508|         // now we've examined, get at least out of
    | this granule
56509|         while ( (CurrentNode!=NULL) &&
    | (CurrentKey.GranulePart == Granule) ) {
56510|             #if DO_ALL_FREESPACE
56511|
    | Debug(DEBUG_DICT,("pd::CreditInterveningGranulesInTree:
    | scope: |%x - %x skipping %8x|%08x - SS too high\n"
56512|                                     , LowerSS
56513|                                     , UpperSS
56514|                                     ,
    | CurrentKey.GranulePart
56515|                                     ,
    | CurrentKey.SnapShotPart
56516|                                     ));
56517|         #endif /*DO_ALL_FREESPACE*/
56518|         StepThroughTree;
56519|     }
56520|
56521| }
56522| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
56523|     Debug(DEBUG_SFILTER,("Exception %08x in
    | CreditInterveningGranulesInTree\n",GetExceptionCode()));
56524| }
56525| }
56526|
56527| //-----
    | -----
56528|
56529| NTSTATUS Rebuild_DeleteJunctionsOnVolume (
    | PDEVICE_OBJECT Volume )
56530| {
56531|     NTSTATUS status = STATUS_SUCCESS;
56532|
    | Debug(DEBUG_DICT,("Rebuild_DeleteJunctionsOnVolume(%08x)
    | \n",Volume));
56533|

```

```

56534|  __try {
56535|      PFILTERED_EXTENSION DevExt =
56536|      | GetFilteredExtension(Volume);
56537|      const ULONG pathChars = 1024;
56538|      WCHAR *path = (WCHAR *)
56539|      | MemAllocateString(pathChars);
56540|      if ( path ) {
56541|          swprintf ( path, L"%s\\%s", DevExt->Name,
56542|          | gSnapshotDirName );
56543|          NTSTATUS tempStatus =
56544|          | SbDeleteAllReparsePointsAndDir (path);
56545|          | Debug(DEBUG_DICT,("Rebuild_DeleteJunctionsOnVolume:
56546|          | after SbDeleteAllReparsePointsAndDir '%S';
56547|          | status=%08x\n",path,tempStatus));
56548|          tempStatus = SbCreateDirectory
56549|          | (path,NULL,FILE_ATTRIBUTE_NORMAL);
56550|          | Debug(DEBUG_DICT,("Rebuild_DeleteJunctionsOnVolume:
56551|          | SbCreateDirectory('%S') returned
56552|          | %08x\n",path,tempStatus));
56553|          MemFreeString(path);
56554|      } else {
56555|          | Debug(DEBUG_DICT,("Rebuild_DeleteJunctionsOnVolume:
56556|          | Out of memory (path)\n"));
56557|          status = STATUS_INSUFFICIENT_RESOURCES;
56558|      }
56559|  } __except(
56560|      | ExceptionFilter(GetExceptionInformation()) ) {
56561|      status = GetExceptionCode();
56562|      Debug(DEBUG_DICT,("!!! Exception %08x in
56563|      | Rebuild_DeleteJunctionsOnVolume;
56564|      | Volume=%08x\n",status,Volume));
56565|  }
56566|  Debug(DEBUG_DICT,("Rebuild_DeleteJunctionsOnVolume
56567|  | returning %08x\n",status));
56568|  return status;
56569| }
56570|
56571| //-----
56572| | -----
56573|
56574| NTSTATUS Rebuild_RecreateJunctionsOnVolume(
56575|     | PDEVICE_OBJECT Volume )
56576| {
56577|     NTSTATUS Status=STATUS_UNSUCCESSFUL;

```

```

56565|
56566|    Debug(DEBUG_DICT,("Entering
    | Rebuild_RecreateJunctionsOnVolume,
    | Vol=%08x\n",Volume));
56567|
56568|    __try {
56569|        PDEVICE_OBJECT DevObj = Volume;
56570|        if (
    | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
56571|            Rebuild_DeleteJunctionsOnVolume (DevObj);
56572|            PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DevObj);
56573|            pkSnapshotEntry
    | p=GetTopSnapshot(&DevExt->Snapshots);
56574|
56575|            while ( p ) {
56576|                PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(p->DeviceObject);
56577|
56578|            | if(p->MasterSnapshot->UserSnapshotName[0]!=0) {
56579|                PDEVICE_OBJECT
    | Virtual=GetVdiskObjectForName(DevExt->Name,p->MasterSnap
    | Shot->Instance);
56580|                if ( Virtual ) {
56581|                    PVDISK_EXTENSION VDisk =
    | GetVDiskExtension(Virtual);
56582|                    WCHAR *linkDirectory = 0;
56583|                    WCHAR *targetDirectory = 0;
56584|
56585|                    linkDirectory =
    | (WCHAR*)MemAllocateString(1024);
56586|                    if(linkDirectory) {
56587|                        targetDirectory =
    | (WCHAR*)MemAllocateString(256);
56588|
56589|                        if(targetDirectory) {
56590|                            swprintf (
    | linkDirectory,L"%s\\%s",DevExt->Name,p->MasterSnapshot->
    | UserSnapshotName );
56591|                            swprintf (
    | targetDirectory,L"%s\\",VDisk->VolumeGuid );
56592|
56593|            | Debug(DEBUG_DICT,("Rebuild_RecreateJunctionsOnVolume:
    | dir='%S'\n",linkDirectory));
56594|
    | Debug(DEBUG_DICT,("Rebuild_RecreateJunctionsOnVolume:
    | target='%S'\n",targetDirectory));
56595|

```

```

    | Debug(DEBUG_DICT,("Rebuild_RecreateJunctionsOnVolume:
    | time=%I64x\n",p->MasterSnapShot->SnapShotTime.QuadPart))
    | ;
56596|         Status = CreateJunction
    | (
    | linkDirectory,targetDirectory,p->MasterSnapShot->SnapSho
    | tTime );
56597|
56598|         if ( NT_SUCCESS(Status)
    | ) {
56599|             | Debug(DEBUG_DICT,("Success creating junction\n"));
56600|             } else {
56601|                 | Debug(DEBUG_DICT,("Error %08x creating
    | junction\n",Status));
56602|                 }
56603|                 | MemFreeString(targetDirectory);
56604|                 } else {
56605|                     | Debug(DEBUG_DICT,("Error out of memory for target\n"));
56606|                     }
56607|                     | MemFreeString(linkDirectory);
56608|                     } else {
56609|                         Debug(DEBUG_DICT,("Error
    | out of memory for link\n"));
56610|                         }
56611|                         } else {
56612|                             Debug(DEBUG_DICT,("Unable to
    | find virtual volume!!!!\n"));
56613|             #ifdef DEBUG
56614|                 DbgBreakPoint();
56615|             #endif
56616|             }
56617|             } else {
56618|                 Debug(DEBUG_DICT,("Error Snapshot
    | has no name\n"));
56619|                 }
56620|                 p =
    | GetNextSnapShot(&DevExt->SnapShots,p);
56621|             }
56622|         }
56623|     } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
56624|         Status = GetExceptionCode();
56625|         Debug(DEBUG_SFILTER,("Exception %08x in
    | Rebuild_RecreateJunctionsOnVolume\n",Status));
56626|     }

```



```

56627|
56628|
56629|   | Debug(DEBUG_DICT,("Rebuild_RecreateJunctionsOnVolume
56630|   | returning %08x\n",Status));
56631|   return Status;
56632| }
56633|
56634| //-----
56635| | -----
56636| | -----
56637| NTSTATUS Rebuild_DismountAllVolumes ( PDEVICE_OBJECT
56638|   | Volume, BOOLEAN DisableMounts )
56639| {
56640|   NTSTATUS Status=STATUS_UNSUCCESSFUL;
56641|   WCHAR Buffer[255];
56642|   Debug(DEBUG_DICT,("Entering
56643|   | Rebuild_DismountAllVolumes, Vol=%08x\n",Volume));
56644|   __try {
56645|       // dismount all volumes
56646|       PDEVICE_OBJECT DevObj = Volume;
56647|       if (
56648|           | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
56649|           PFILTERED_EXTENSION DevExt =
56650|           | GetFilteredExtension(DevObj);
56651|           pkSnapshotEntry
56652|           | p=GetTopSnapshot(&DevExt->Snapshots);
56653|           while ( p ) {
56654|               PFILTERED_EXTENSION DevExt =
56655|               | GetFilteredExtension(p->DeviceObject);
56656|               PDEVICE_OBJECT
56657|               | Virtual=GetVdiskObjectForName(DevExt->Name,p->MasterSnap
56658|               | Shot->Instance);
56659|               if ( Virtual ) {
56660|                   PVDISK_EXTENSION VDisk =
56661|                   | GetVDiskExtension(Virtual);
56662|                   |
56663|                   | swprintf(Buffer,L"\\Device\\PsmDevices_\\%04x\\%s_\\%d",PSM_
56664|                   | LOW_COMPATIBLE_VERSION,VDisk->Name,p->MasterSnapshot->In
56665|                   | stance);
56666|                   Debug(DEBUG_DICT,("Dismounting
56667|                   | volume '%S'\n",Buffer));
56668|                   if(DisableMounts) {
56669|                       VDisk->MountDisabled=TRUE;
56670|                   }
56671|                   Status = Sblo_DismountVolume(

```

```

    | Buffer );
56660|         if ( NT_SUCCESS(Status) ) {
56661|             Debug(DEBUG_DICT,("Success
    | dismounting volume\n"));
56662|         } else {
56663|             Debug(DEBUG_DICT,("Error %08x
    | dismounting volume\n",Status));
56664|         }
56665|
56666|     } else {
56667|         // This can happen before stage 2
    | had a chance to run or complete.
56668|         Debug(DEBUG_DICT,("Unable to find
    | virtual volume!!!!\n"));
56669|     }
56670|     p =
    | GetNextSnapShot(&DevExt->SnapShots,p);
56671|     }
56672|     }
56673| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
56674|     Status = GetExceptionCode();
56675|     Debug(DEBUG_SFILTER,("Exception %08x in
    | dismount all volumes\n",Status));
56676| }
56677|
56678| Debug(DEBUG_DICT,("Rebuild_DismountAllVolumes
    | returning %08x\n",Status));
56679| return Status;
56680| }
56681|
56682| //-----
    | -----
    | -----
56683| NTSTATUS Rebuild_ReenableVolumeMounts ( PDEVICE_OBJECT
    | Volume )
56684| {
56685|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
56686|
56687|     Debug(DEBUG_DICT,("Entering
    | Rebuild_ReenableVolumeMounts , Vol=%08x\n",Volume));
56688|
56689|     __try {
56690|         // reenale mounting of all virtual volumes
56691|         PDEVICE_OBJECT DevObj = Volume;
56692|         if (
    | PsmGetObjectype(DevObj)==OBJECT_FILTEREDDISK ) {
56693|             PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(DevObj);
56694|             pkSnapShotEntry

```

```

    | p=GetTopSnapShot(&DevExt->SnapShots);
56695|
56696|         while ( p ) {
56697|             PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(p->DeviceObject);
56698|             PDEVICE_OBJECT
    | Virtual=GetVdiskObjectForName(DevExt->Name,p->MasterSnap
    | Shot->Instance);
56699|             if ( Virtual ) {
56700|                 PVDISK_EXTENSION VDisk =
    | GetVDiskExtension(Virtual);
56701|
56702|                 VDisk->MountDisabled=FALSE;
56703|             } else {
56704|                 // This can happen before stage 2
    | had a chance to run or complete.
56705|                 Debug(DEBUG_DICT,("Unable to find
    | virtual volume!!!!\n"));
56706|             }
56707|             p =
    | GetNextSnapShot(&DevExt->SnapShots,p);
56708|         }
56709|     }
56710| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
56711|     Status = GetExceptionCode();
56712|     Debug(DEBUG_SFILTER,("Exception %08x in
    | reenable volume mounts\n",Status));
56713| }
56714|
56715| Debug(DEBUG_DICT,("Rebuild_ReenableVolumeMounts
    | returning %08x\n",Status));
56716| return Status;
56717| }
56718|
56719| //-----
    | -----
    | -----
56720|
56721| NTSTATUS Rebuild_MountAllVolumes ( PDEVICE_OBJECT
    | Volume )
56722| {
56723|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
56724|     WCHAR Buffer[255];
56725|
56726|     Debug(DEBUG_DICT,("Entering
    | Rebuild_MountAllVolumes, vol=%08x\n",Volume));
56727|
56728|     __try {
56729|         PDEVICE_OBJECT DevObj = Volume;

```

```

56730|         if (
| PsmGetObjectype(DevObj)==OBJECT_FILTEREDDISK ) {
56731|             PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(DevObj);
56732|
56733|             pkSnapshotEntry
| p=GetTopSnapshot(&DevExt->Snapshots);
56734|
56735|             while ( p ) {
56736|                 PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(p->DeviceObject);
56737|                 PDEVICE_OBJECT
| Virtual=GetVdiskObjectForName(DevExt->Name,p->MasterSnap
| Shot->Instance);
56738|                 if ( Virtual ) {
56739|                     PVDISK_EXTENSION VDisk =
| GetVDiskExtension(Virtual);
56740|
56741|                     | swprintf(Buffer,L"\\Device\\PsmDevices_%%04x\\%%s_%%d",PSM_
| LOW_COMPATIBLE_VERSION,VDisk->Name,p->MasterSnapshot->In
| stance);
56742|
56743|                     // Make this disk a Unique
| Number... This is to cause the file system to
56744|                     // discard any cached data for the
| old volume (due to a forced dismount)
56745|                     // this however will cause all
| volumes to go through the full mount stage
56746|                     // even if the volume was not
| forced shutdown
56747|
56748|                     LARGE_INTEGER BO;
56749|                     /*lint -save -e740 */
56750|                     KeQueryTickCount( &BO );
56751|                     /*lint -restore */
56752|                     VDisk->SerialNumber = BO.LowPart;
56753|
56754|                     Debug(DEBUG_DICT,("Mounting volume
| '%S\\n",Buffer));
56755|                     Status = SbTouchVolume( Buffer );
56756|                     if ( NT_SUCCESS(Status) ) {
56757|                         Debug(DEBUG_DICT,("Success
| mounting volume\\n"));
56758|
56759|                         // lets delete the snapshot
| directory off of the virtual drive
56760|                         // so nesting will not occur
| when other snapshots exist
56761|

```

```

56762|         wscat(Buffer,L"\\");
56763|
56764|         | wscat(Buffer,gSnapShotDirName);
56764|         Debug(DEBUG_DEVSUP,("Deleting
56764|         | directory '%S\\n",Buffer));
56765|         SbSnapShotCleanup(Buffer);
56766|
56767|         } else {
56768|         Debug(DEBUG_DICT,("Error %08x
56768|         | mounting volume\\n",Status));
56769|         }
56770|
56771|         } else {
56772|         // This can happen before stage 2
56772|         | had a chance to run or complete.
56773|         Debug(DEBUG_DICT,("Unable to find
56773|         | virtual volume.\\n"));
56774|         }
56775|         p =
56775|         | GetNextSnapShot(&DevExt->SnapShots,p);
56776|         }
56777|         }
56778|     } __except(
56778|         | ExceptionFilter(GetExceptionInformation()) ) {
56779|         Status = GetExceptionCode();
56780|         Debug(DEBUG_SFILTER,("Exception %08x in Mount
56780|         | all volumes\\n",Status));
56781|     }
56782|
56783|     Debug(DEBUG_DICT,("Rebuild_MountAllVolumes
56783|     | returning %08x\\n",Status));
56784|     return Status;
56785|
56786| }
56787|
56788| //-----
56788| | -----
56789|
56790| WCHAR *GetPrefix (
56791|     const WCHAR * const    Location,
56792|     WCHAR *                Prefix )
56793| {
56794|     // determine where in the object name space the
56795|     // locations are
56796|     // C:\      d
56797|     if ( Location[1]==L':' ) {
56798|         // drive letter passed in
56799|         wcscpy(Prefix,L"\\DosDevices\\");
56800|     } else
56801|         // Volume{

```

```

56802|     if ( (Location[0]==L'V') && (Location[6]==L'{')
56803|         | ) {
56804|         wcsncpy(Prefix,L"\\?\\");
56805|     } else {
56806|         if ( Location[0]==L'\\' ) {
56807|             // already a valid path
56808|             wcsncpy(Prefix,L "");
56809|         } else {
56810|             Debug(DEBUG_DEVCON,("Invalid object name
56811| | space name '%S\\n",Location));
56812|             // invalid name
56813|             ASSERT(FALSE);
56814|             wcsncpy(Prefix,L "");
56815|         }
56816|     }
56817|     return Prefix;
56818| }
56819|
56820| //-----
56821| | -----
56822| #if 1
56823| NTSTATUS AddPSMDirToDoNotBackupList( )
56824| {
56825|     ULONG Len=256;
56826|     WCHAR *DirToAdd=(WCHAR*)MemAllocateString(Len);
56827|     NTSTATUS Status;
56828|     if(DirToAdd) {
56829|         wcsncpy(DirToAdd,L "\\");
56830|         wscat(DirToAdd,PSM_PRIVATE_DIR_SLASH);
56831|         wscat(DirToAdd,L "*/s");
56832|         Len = wcslen(DirToAdd);
56833|         DirToAdd[Len] = 0;
56834|         DirToAdd[Len+1] = 0;
56835|         Status = RtlWriteRegistryValue(
56836|             RTL_REGISTRY_ABSOLUTE,
56837|             L"\\Registry\\Machine\\SYSTEM\\CurrentControlSet\\Contro
56838|             | \\BackupRestore\\FilesNotToBackup",
56839|             L"Persistent Storage Manager (Files)",
56840|             REG_MULTI_SZ,
56841|             DirToAdd,
56842|             (Len*sizeof(WCHAR))+(sizeof(WCHAR)*2));
56843|         Debug(DEBUG_DICT,("AddPSMDirToDoNotBackupList:
56844| | Status=%08x\\n",Status));
56845|         MemFreeString(DirToAdd);
56846|     } else {

```

```

56846|     Status = STATUS_INSUFFICIENT_RESOURCES;
56847| }
56848| return Status;
56849| }
56850|
56851| //-----
| -----
56852|
56853| NTSTATUS StoreFileSystem( PDEVICE_OBJECT Volume, HANDLE
| Handle )
56854| {
56855|     NTSTATUS Status;
56856|     PFILE_OBJECT FileObject;
56857|     ULONG FileSystem;
56858|     PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
56859|     WCHAR Buffer[255]={0};
56860|     PFILE_FS_ATTRIBUTE_INFORMATION
| Attrib=(PFILE_FS_ATTRIBUTE_INFORMATION)Buffer;
56861|
56862|     PAGED_CODE();
56863|
56864|     Status = ObReferenceObjectByHandle(
56865|         Handle,      // IN HANDLE Handle,
56866|         0, // IN ACCESS_MASK DesiredAccess,
56867|         NULL,        // IN POBJECT_TYPE
| ObjectType,        // optional
56868|         (KPROCESSOR_MODE)KernelMode,      // IN
| KPROCESSOR_MODE AccessMode,
56869|         (PVOID*)&FileObject,      // OUT PVOID
| *Object,
56870|         NULL          // OUT
| POBJECT_HANDLE_INFORMATION HandleInformation //
| optional
56871|     );
56872|     if(NT_SUCCESS(Status)) {
56873|         Status = FS_GetVolumeAttributes( FileObject,
| Attrib, sizeof(Buffer) );
56874|         if(NT_SUCCESS(Status)) {
56875|             // null terminate
56876|
| Attrib->FileSystemName[Attrib->FileSystemNameLength/2]=0
| ;
56877|
| if(_wcsicmp(Attrib->FileSystemName,L"NTFS")==0) {
56878|             FileSystem = FILE_SYSTEM_NTFS;
56879|         } else
56880|
| if(_wcsicmp(Attrib->FileSystemName,L"FAT")==0) {
56881|             FileSystem = FILE_SYSTEM_FAT;

```

```

56882|         } else {
56883|             FileSystem = FILE_SYSTEM_UNKNOWN;
56884|         }
56885|
56886|         // store the file system type in the per
        | volume area of the registry
56887|         WCHAR *VolumeReg =
        | GetPerVolumeRegistry(Volume);
56888|         if(VolumeReg) {
56889|             | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,VolumeReg,L"
        | FileSystem",REG_DWORD,&FileSystem,sizeof(DWORD));
56890|             DevExt->FileSystem = FileSystem;
56891|             FreePerVolumeRegistry(VolumeReg);
56892|         }
56893|     } else {
56894|         Debug(DEBUG_DICT,("Error %08x getting
        | volume attributes\n",Status));
56895|     }
56896|     ObDereferenceObject (FileObject);
56897| } else {
56898|     Debug(DEBUG_DICT,("Error %08x getting file
        | object for file handle\n",Status));
56899| }
56900| return Status;
56901| }
56902|
56903|
56904| NTSTATUS PersistentDictionary::CreateFilesForVolume(
        | PDEVICE_OBJECT Volume, PVOID AbortEvent )
56905| {
56906|     NTSTATUS Status = STATUS_UNSUCCESSFUL;
56907|     WCHAR Header[300];
56908|     WCHAR Cache[300];
56909|     WCHAR Index[300];
56910|     LARGE_INTEGER Size;
56911|     PFILTERED_EXTENSION DevExt =
        | GetFilteredExtension(Volume);
56912|
56913|     UpdateGlobalStatus(PSM_CREATING_FILES);
56914|
        | Reg_GetULONGKey(&gRegistryPath,L"FillOnWrite",FILLONWRIT
        | ES_DEF,&PSManFillOnWrite);
56915|
56916|     Debug(DEBUG_DICT,("pd::CreateFilesForVolume:
        | Volume=%08x, uni='%S',
        | guid='%S'\n",Volume,DevExt->UniqueId,DevExt->VolumeGuid)
        | );
56917|     __try {
56918|         PSECURITY_DESCRIPTOR SD = SbGetAdminOnlySD();

```



```

56919|
56920|     | GetPrefix(DevExt->Cache.HeaderFile.Location,Header);
56921|
56922|     | wcscat(Header,DevExt->Cache.HeaderFile.Location);
56923|     | wcscat(Header,PSM_PRIVATE_DIR_SLASH);
56924|
56925|     | SbCreateDirectory(Header,SD,FILE_ATTRIBUTE_HIDDEN);
56926|
56927|     if(DevExt->Uniqueld[0]) {
56928|         | wcscat(Header,DevExt->Uniqueld);
56929|         | wcscat(Header,L".header.psm");
56930|     } else
56931|     if(DevExt->VolumeGuid[0]) {
56932|         | wcscat(Header,DevExt->VolumeGuid);
56933|         | wcscat(Header,L".header.psm");
56934|     } else {
56935|         | wcscat(Header,L"header.psm");
56936|     }
56937|
56938|     | GetPrefix(DevExt->Cache.IndexFile.Location,Index);
56939|     | wcscat(Index,DevExt->Cache.IndexFile.Location);
56940|     | wcscat(Index,PSM_PRIVATE_DIR_SLASH);
56941|
56942|     | SbCreateDirectory(Index,SD,FILE_ATTRIBUTE_HIDDEN);
56943|
56944|     if(DevExt->Uniqueld[0]) {
56945|         | wcscat(Index,DevExt->Uniqueld);
56946|         | wcscat(Index,L".index.psm");
56947|     } else
56948|     if(DevExt->VolumeGuid[0]) {
56949|         | wcscat(Index,DevExt->VolumeGuid);
56950|         | wcscat(Index,L".index.psm");
56951|     } else {
56952|         | wcscat(Index,L"index.psm");
56953|     }
56954|
56955|     | GetPrefix(DevExt->Cache.CacheFile.Location,Cache);
56956|     | wcscat(Cache,DevExt->Cache.CacheFile.Location);
56957|     | wcscat(Cache,PSM_PRIVATE_DIR_SLASH);
56958|
56959|     | SbCreateDirectory(Cache,SD,FILE_ATTRIBUTE_HIDDEN);
56960|
56961|     if(DevExt->Uniqueld[0]) {
56962|         | wcscat(Cache,DevExt->Uniqueld);
56963|         | wcscat(Cache,L".diff.psm");
56964|     } else
56965|     if(DevExt->VolumeGuid[0]) {
56966|         | wcscat(Cache,DevExt->VolumeGuid);
56967|         | wcscat(Cache,L".diff.psm");
56968|     } else {
56969|         | wcscat(Cache,L"diff.psm");
56970|     }

```

```

56962|     } else {
56963|         wscat(Cache,L"diff.psm");
56964|     }
56965|
56966|     // free security descriptor
56967|     if(SD) {
56968|         MemFreePool(SD);
56969|     }
56970|
56971|     AddPSMDirToDoNotBackupList();
56972|
56973|     Size.QuadPart =
56974|         sizeof(tHeader) +
56975|         | MAX_SNAPSHOTS_PER_HEADER_FILE*sizeof(tDiskInternalSnapSh
56976|         | ot) +
56977|         sizeof(DWORD);
56978|     Size.QuadPart = ROUND_UP(Size.QuadPart,
56979|         | DevExt->BytesPerSector);
56980|     Size.QuadPart *= NumSavedHeaderBlocks;
56981|     Debug(DEBUG_DICT,("pd::CreateFilesForVolume:
56982|     | Header Size = %08l64x, Release 2 was
56983|     | %08x\n",Size.QuadPart,HEADER_SIZE_IN_RELEASE_2));
56984|     ASSERT(Size.QuadPart <=
56985|         | HEADER_SIZE_IN_RELEASE_2);
56986|     Size.QuadPart = HEADER_SIZE_IN_RELEASE_2; //
56987|     | maintain backward compatibility
56988|
56989|     Status =
56990|         | SbCreateAndFillFile(Header,&Size,AbortEvent,0x00,FILLONW
56991|         | RITE_ALL);
56992|     if ( NT_SUCCESS(Status) ) {
56993|         Size = RtlEnlargedUnsignedMultiply (
56994|         | DevExt->Cache.PSManBitMapSize, DevExt->BytesPerSector
56995|         | );
56996|         Status =
56997|         | SbCreateAndFillFile(Index,&Size,AbortEvent,0x00,FILLONWR
56998|         | ITE_ALL);
56999|
57000|     if ( NT_SUCCESS(Status) ) {
57001|         HANDLE TempHandle =
57002|         | INVALID_HANDLE_VALUE;
57003|         PFILE_OBJECT TempObject = NULL;
57004|         HANDLE TempWaitHandle =
57005|         | INVALID_HANDLE_VALUE;
57006|         PVOID TempWaitObject = NULL;
57007|
57008|         Size = RtlEnlargedUnsignedMultiply (
57009|         | DevExt->Cache.PSManBitMapSize, GRANULE_SIZE);

```

```

56996|
56997|          // since the cache file can get huge,
    | lets
56998|          // reuse it whenever we can instead of
    | recreating it
56999|          Status =
    | OpenAFile(Cache,TempHandle,TempObject,TempWaitHandle,Temp
    | pWaitObject);
57000|          if ( NT_SUCCESS(Status) ) {
57001|              StoreFileSystem(Volume,TempHandle);
57002|
    | CloseAFile(TempHandle,TempObject,TempWaitHandle,TempWait
    | Object);
57003|              try_return(NOTHING);
57004|          } else {
57005|              Status =
    | SbCreateAndFillFile(Cache,&Size,AbortEvent,0x00,PSManFil
    | IOnWrite);
57006|              if ( NT_SUCCESS(Status) ) {
57007|                  // store what file system is
    | being used for this volume
57008|                  Status =
    | OpenAFile(Cache,TempHandle,TempObject,TempWaitHandle,Temp
    | pWaitObject);
57009|                  if ( NT_SUCCESS(Status) ) {
57010|                      StoreFileSystem(Volume,TempHandle);
57011|                      CloseAFile(TempHandle,TempObject,TempWaitHandle,TempWait
    | Object);
57012|                  } else {
57013|                      // odd, unable to open file
    | we just created successfully
57014|                  }
57015|                  try_return(NOTHING);
57016|              } else {
57017|
    | Debug(DEBUG_DICT,("CreateFilesForVolume: Error %08x
    | creating cache\n",Status));
57018|              }
57019|          }
57020|          SbDeleteFile(Index,
    | FILE_ATTRIBUTE_NORMAL);
57021|      } else {
57022|
    | Debug(DEBUG_DICT,("CreateFilesForVolume: Error %08x
    | creating index\n",Status));
57023|      }
57024|      SbDeleteFile(Header,
    | FILE_ATTRIBUTE_NORMAL);

```

```

57025|     } else {
57026|         Debug(DEBUG_DICT,("CreateFilesForVolume:
| Error %08x creating header\n",Status));
57027|     }
57028|     try_exit:NOTHING;
57029| } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
57030|     Status = GetExceptionCode();
57031|     Debug(DEBUG_DICT,("Exception %08x in
| pd::CreateFilesForVolume\n",Status));
57032| }
57033|
57034|     Debug(DEBUG_DICT,("pd::CreateFilesForVolume
| returning %08x\n",Status));
57035|     return Status;
57036| }
57037|
57038| NTSTATUS
57039| PersistentDictionary::CloseFilesForVolume(
| PDEVICE_OBJECT Volume )
57040| {
57041|     PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
57042|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
57043|
57044|     //Debug(DEBUG_DICT,("pd::CloseFilesForVolume:
| Volume=%08x\n",Volume));
57045|     __try {
57046|         CloseProcessHandle
| (DevExt->Cache.CacheFile.FileHandle, (PVOID
| &)(DevExt->Cache.CacheFile.FileObject) );
57047|         CloseProcessHandle
| (DevExt->Cache.IndexFile.FileHandle, (PVOID
| &)(DevExt->Cache.IndexFile.FileObject) );
57048|         CloseProcessHandle
| (DevExt->Cache.HeaderFile.FileHandle, (PVOID
| &)(DevExt->Cache.HeaderFile.FileObject) );
57049|
57050|         CloseProcessHandle
| (DevExt->Cache.CacheFile.WaitHandle,
| DevExt->Cache.CacheFile.WaitObject);
57051|         CloseProcessHandle
| (DevExt->Cache.IndexFile.WaitHandle,
| DevExt->Cache.IndexFile.WaitObject);
57052|         CloseProcessHandle
| (DevExt->Cache.HeaderFile.WaitHandle,
| DevExt->Cache.HeaderFile.WaitObject);
57053|         Status = STATUS_SUCCESS;
57054|     } __except(
| ExceptionFilter(GetExceptionInformation()) ) {

```

```

57055|     Status = GetExceptionCode();
57056|     Debug(DEBUG_DICT,("Exception %08x in
| pd::CloseFilesForVolume\n",Status));
57057| }
57058|
57059|     Debug(DEBUG_DICT,("pd::CloseFilesForVolume %08x
| returning %08x\n",Volume,Status));
57060|     return Status;
57061| }
57062|
57063| #define abs(x) ((x) >=0 ? (x) : -(x))
57064|
57065| //-----
| -----
57066|
57067| void GetCacheSizes( PFILTERED_EXTENSION DevExt, ULONG
| &InitialSize, ULONG &MaxSize )
57068| {
57069|     __try {
57070|         signed long IS=(signed
| long)DevExt->Cache.InitialSize;
57071|         signed long MS=(signed
| long)DevExt->Cache.MaxSize;
57072|         ULONG ISC=0;
57073|         ULONG MSC=0;
57074|         LARGE_INTEGER L = {0};
57075|         NTSTATUS Status;
57076|
57077|         InitialSize = 0;
57078|         MaxSize = 0;
57079|
57080|         // Before sizes stored in DevExt->Cache, make
| sure they are up to date.
57081|         // Call pd::GetStateForVolume to refresh latest
| values from registry.
57082|         PDEVICE_OBJECT Volume = DevExt->DeviceObject;
57083|         ASSERT(GetFilteredExtension(Volume)==DevExt);
57084|         PersistentDictionary::GetStateForVolume
| (Volume);
57085|
57086|         if(DevExt->PSMed) {
57087|             // volume already has a cache file. lets
| use its size
57088|             Status =
| DevExt->Cache.CacheFile.Direct->getFileSizeInBytes(L);
57089|
57090|             if(NT_SUCCESS(Status)) {
57091|                 ASSERT(L.QuadPart);
| // make sure not zero
57092|                 ASSERT((L.QuadPart % (1024*1024)) ==

```

```

    | 0); // make sure it is an integer number of
    | megabytes
57093|         // convert to number of megabytes
57094|         L.QuadPart/=(1024*1024);
57095|         // cant handle things this large
57096|         ASSERT(L.HighPart==0);
57097|         // set the max and init size to the
    | same for now
57098|         InitialSize = MaxSize = L.LowPart;
57099|         Debug(DEBUG_DICT,("GetCacheSizes:
    | Reusing cache file size: cache file is %d
    | MB\n",InitialSize));
57100|     } else {
57101|         Debug(DEBUG_DICT,("GetCacheSizes: Error
    | %08x getting cache file size\n",Status));
57102|         goto Calculatelt;
57103|     }
57104| } else {
57105|     // volume has no snapshots, so calculate
    | the size incase it changed
57106|     // from the last time the cache file was
    | created
57107| Calculatelt:
57108|     // if number is less than 0, then it is in
    | percentage
57109|     if(IS<0) {
57110|         // how many MB
57111|         L.QuadPart =
    | DevExt->Pi.PartitionLength.QuadPart / (1024*1024);
57112|         ASSERT(L.HighPart == 0);
57113|         // get percentage of space in MB
57114|         ISC = (L.LowPart * (-IS)) / 100;
57115|         Debug(DEBUG_DICT,("GetCacheSizes: Init:
    | Volume is %d MB, cache= %d%%, (%d
    | MB)\n",L.LowPart,abs(IS),ISC));
57116|     } else {
57117|         ISC = (ULONG)IS;
57118|     }
57119|
57120|     if(MS<0) {
57121|         // how many MB
57122|         L.QuadPart =
    | DevExt->Pi.PartitionLength.QuadPart / (1024*1024);
57123|         ASSERT(L.HighPart == 0);
57124|         // get percentage of space in MB
57125|         MSC = (L.LowPart * (-MS)) / 100;
57126|         Debug(DEBUG_DICT,("GetCacheSizes: Max:
    | Volume is %d MB, cache= %d%%, (%d
    | MB)\n",L.LowPart,abs(MS),MSC));
57127|     } else {

```

```

57128|         MSC = (ULONG)MS;
57129|     }
57130|
57131|     InitialSize = ISC;
57132|     MaxSize = MSC;
57133| } // PSMed
57134|
57135| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
57136|     NTSTATUS Status = GetExceptionCode();
57137|     Debug(DEBUG_DICT,("!!! Exception %08x in
    | GetCacheSizes\n",Status));
57138| }
57139| }
57140|
57141| //-----
    | -----
57142|
57143| NTSTATUS
    | PersistentDictionary::InitializeFileNamesForVolume (
    | PDEVICE_OBJECT Volume )
57144| {
57145|     NTSTATUS Status = STATUS_SUCCESS;
57146|
57147|     __try {
57148|         PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
57149|
57150|         WCHAR Dir[300];
57151|         wcscpy(Dir,PSM_PRIVATE_DIR_SLASH);
57152|
57153|         if(DevExt->Uniqueld[0]) {
57154|             wcscat(Dir,DevExt->Uniqueld);
57155|             wcscat(Dir,L".");
57156|         } else if(DevExt->VolumeGuid[0]) {
57157|             wcscat(Dir,DevExt->VolumeGuid);
57158|             wcscat(Dir,L".");
57159|         } else {
57160|             // Just append short file name later
57161|         }
57162|
57163|         const WCHAR * const FileTable[] = {
57164|             // short filename      location
    | where to put complete name
57165|             L"header.psm",
    | DevExt->Cache.HeaderFile.Location,
    | DevExt->Cache.HeaderFile.FileName,
57166|             L"index.psm",
    | DevExt->Cache.IndexFile.Location,
    | DevExt->Cache.IndexFile.FileName,

```

```

57167|         L"diff.psm",
      | DevExt->Cache.CacheFile.Location,
      | DevExt->Cache.CacheFile.FileName,
57168|         NULL,         NULL,
      | NULL
57169|     };
57170|
57171|     for ( ULONG i=0; FileTable[i] != NULL; i+=3 ) {
57172|         const WCHAR * const   ShortFileName =
      | FileTable[i];
57173|         const WCHAR * const   Location      =
      | FileTable[i+1];
57174|         WCHAR *              Destination    =
      | (WCHAR *) FileTable[i+2];
57175|
57176|         ASSERT(Location != NULL);
57177|         ASSERT(Destination != NULL);
57178|
57179|         if ( Destination[0] == L'\0' ) {
57180|             GetPrefix (Location, Destination);
57181|             wcscat (Destination, Location);
57182|             wcscat (Destination, Dir);
57183|             wcscat (Destination, ShortFileName);
57184|         } else {
57185|             | Debug(DEBUG_DICT,("pd::InitializeFileNamesForVolume:
      | Filename for %S was already defined - leaving
      | alone\n",ShortFileName));
57186|         }
57187|
57188|             | Debug(DEBUG_DICT,("pd::InitializeFileNamesForVolume:
      | Info for FileTable[%d] ...\n",i));
57189|             Debug(DEBUG_DICT,("  Short='%S'
      | ...\n",ShortFileName));
57190|             Debug(DEBUG_DICT,("  Loc=
      | '%S'\n",Location));
57191|             Debug(DEBUG_DICT,("  Dest=
      | '%S'\n",Destination));
57192|         }
57193|     } __except(
      | ExceptionFilter(GetExceptionInformation()) ) {
57194|         Status = GetExceptionCode();
57195|         Debug(DEBUG_DICT,("!!! Exception %08x in
      | pd::InitializeFileNamesForVolume\n",Status));
57196|     }
57197|
57198|     return Status;
57199| }
57200|

```



```

57201| //-----
| -----
57202|
57203| NTSTATUS PersistentDictionary::OpenFilesForVolume (
| PDEVICE_OBJECT Volume )
57204| {
57205|     NTSTATUS Status = STATUS_UNSUCCESSFUL;
57206|     BOOLEAN NewFiles=FALSE;
57207|     PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
57208|
57209|     // dont check for this here, as this routine can be
| called from
57210|     // other cases than just create snapshot (as in
| create files)
57211|     // ASSERT(GlobalSystemProcessId ==
| PsGetCurrentProcess());
57212|     Debug(DEBUG_DICT,("pd::OpenFilesForVolume:
| Volume=%08x, uni='%S',
| guid='%S\\n", Volume, DevExt->UniquelId, DevExt->VolumeGuid)
| );
57213|     __try {
57214|         InitializeFileNamesForVolume (Volume);
57215|
57216|         Status = OpenAFile (
57217|             DevExt->Cache.HeaderFile.FileName,
57218|             DevExt->Cache.HeaderFile.FileHandle,
57219|             DevExt->Cache.HeaderFile.FileObject,
57220|             DevExt->Cache.HeaderFile.WaitHandle,
57221|             DevExt->Cache.HeaderFile.WaitObject );
57222|
57223|         if ( NT_SUCCESS(Status) ) {
57224|             Status = OpenAFile (
57225|                 DevExt->Cache.IndexFile.FileName,
57226|                 DevExt->Cache.IndexFile.FileHandle,
57227|                 DevExt->Cache.IndexFile.FileObject,
57228|                 DevExt->Cache.IndexFile.WaitHandle,
57229|                 DevExt->Cache.IndexFile.WaitObject );
57230|
57231|             if ( NT_SUCCESS(Status) ) {
57232|                 Status = OpenAFile (
57233|                     DevExt->Cache.CacheFile.FileName,
57234|                     DevExt->Cache.CacheFile.FileHandle,
57235|                     DevExt->Cache.CacheFile.FileObject,
57236|                     DevExt->Cache.CacheFile.WaitHandle,
57237|                     DevExt->Cache.CacheFile.WaitObject
| );
57238|
57239|                 if ( NT_SUCCESS(Status) ) {
57240|                     FILE_STANDARD_INFORMATION

```

```

    | FSInfo={0};
57241|         IO_STATUS_BLOCK IoStatus;
57242|
57243|         Status = ZwQueryInformationFile(
57244|         | DevExt->Cache.CacheFile.FileHandle,    // IN HANDLE
    | FileHandle,
57245|         &IoStatus,
    | // OUT PIO_STATUS_BLOCK IoStatusBlock,
57246|         &FSInfo,
    | // IN PVOID FileInformation,
57247|         | sizeof(FILE_STANDARD_INFORMATION),    // IN ULONG
    | Length,
57248|         FileStandardInformation );
    | // IN FILE_INFORMATION_CLASS FileInformationClass
57249|
57250|         if(NT_SUCCESS(Status)) {
57251|             // if snapshots exist, dont
    | look at the file
57252|             // sizes
57253|             if(!DevExt->PSMed) {
57254|                 ULONG ISC, MSC;
57255|                 GetCacheSizes( DevExt, ISC,
    | MSC );
57256|
57257|                 // ISC is in megabytes
57258|                 // EndOfFile is in bytes
57259|
57260|
    | if((FSInfo.EndOfFile.QuadPart/1024/1024) == ISC) {
57261|
    | Debug(DEBUG_DICT,("OpenFilesForVolume: Files are okay,
    | using them\n"));
57262|
    | PersistentDictionary::StoreClustersOfFiles(Volume);
57263|         try_return(NOTHING);
57264|     } else {
57265|
    | FILE_DISPOSITION_INFORMATION Del={0};
57266|
57267|
    | Debug(DEBUG_DICT,("OpenFilesForVolume: File size doesnt
    | match, recreating files\n"));
57268|
57269|         Del.DeleteFile = TRUE;
57270|
57271|         ZwSetInformationFile(
    | DevExt->Cache.CacheFile.FileHandle ,&IoStatus, &Del,
    | sizeof(Del),FileDispositionInformation);

```

```

57272|                ZwSetInformationFile(
| DevExt->Cache.IndexFile.FileHandle ,&IoStatus, &Del,
| sizeof(Del),FileDispositionInformation);
57273|                ZwSetInformationFile(
| DevExt->Cache.HeaderFile.FileHandle,&IoStatus, &Del,
| sizeof(Del),FileDispositionInformation);
57274|                Status =
| STATUS_OBJECT_NAME_NOT_FOUND;
57275|                }
57276|                } else {
57277|                | Debug(DEBUG_DICT,("OpenFilesForVolume: Snapshots exist,
| using existing files\n"));
57278|                try_return(NOTHING);
57279|                }
57280|                } else {
57281|                | Debug(DEBUG_DICT,("OpenFilesForVolume: Error %08x
| getting file size\n",Status));
57282|                }
57283|
57284|                CloseAFile (
57285|                | DevExt->Cache.CacheFile.FileHandle,
57286|                | DevExt->Cache.CacheFile.FileObject,
57287|                | DevExt->Cache.CacheFile.WaitHandle,
57288|                | DevExt->Cache.CacheFile.WaitObject );
57289|                } else {
57290|                | Debug(DEBUG_DICT,("OpenFilesForVolume: Error %08x
| opening diff file\n",Status));
57291|                DevExt->Cache.CacheFile.WaitObject
| = DevExt->Cache.CacheFile.FileObject = NULL;
57292|                DevExt->Cache.CacheFile.WaitHandle
| = DevExt->Cache.CacheFile.FileHandle =
| INVALID_HANDLE_VALUE;
57293|                }
57294|
57295|                CloseAFile (
57296|                DevExt->Cache.IndexFile.FileHandle,
57297|                DevExt->Cache.IndexFile.FileObject,
57298|                DevExt->Cache.IndexFile.WaitHandle,
57299|                DevExt->Cache.IndexFile.WaitObject
| );
57300|
57301|                } else {
57302|                Debug(DEBUG_DICT,("OpenFilesForVolume:

```

```

    | Error %08x opening index file\n",Status));
57303|     }
57304|
57305|     CloseAFile (
57306|         DevExt->Cache.HeaderFile.FileHandle,
57307|         DevExt->Cache.HeaderFile.FileObject,
57308|         DevExt->Cache.HeaderFile.WaitHandle,
57309|         DevExt->Cache.HeaderFile.WaitObject );
57310|     } else {
57311|         Debug(DEBUG_DICT,("OpenFilesForVolume:
    | Error %08x opening volume header file\n",Status));
57312|     }
57313|     try_exit:NOTHING;
57314| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
57315|     Status = GetExceptionCode();
57316|     Debug(DEBUG_DICT,("Exception %08x in
    | pd::OpenFilesForVolume\n",Status));
57317| }
57318|
57319|     Debug(DEBUG_DICT,("pd::OpenFilesForVolume returning
    | %08x\n",Status));
57320|     return Status;
57321| }
57322|
57323| //-----
    | -----
57324|
57325| ULONG ConvertNodesToMegs ( ULONG NumNodes )
57326| {
57327|     // Round up to integer number of megabytes.
57328|     const ULONG NumMegs = (NumNodes * sizeof(tTreeLeaf)
    | + 1024*1024-1) / (1024*1024);
57329|     Debug(DEBUG_MEMORY,("ConvertNodesToMegs: given
    | NumNodes=%08x, calculated NumMegs=%08x\n", NumNodes,
    | NumMegs));
57330|     return NumMegs;
57331| }
57332|
57333| //-----
    | -----
57334|
57335| NTSTATUS PersistentDictionary::AllocMemoryForVolume(
    | PDEVICE_OBJECT Volume, BOOLEAN UseDefaults )
57336| {
57337|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
57338|     PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
57339|
57340|     Debug(DEBUG_DICT,("pd::AllocMemoryForVolume:

```

```

    | Volume=%08x, Use=%d\n", Volume, UseDefaults));
57341|
57342|   UpdateGlobalStatus(PSM_RESOURCE_ACQUISITION);
57343|   __try {
57344|       ULONG ISC=0;
57345|       ULONG MSC=0;
57346|
57347|       GetCacheSizes( DevExt, ISC, MSC );
57348|
57349|       // mutex for bitmap functions
57350|
    | ExInitializeFastMutex(&DevExt->Cache.PSManBitMapMutex);
57351|
57352|       // make sure granule size isnt something odd
57353|       ASSERT((1024*1024) % GRANULE_SIZE==0);
57354|       ASSERT(ISC);
57355|       ASSERT(MSC);
57356|
57357|       if(UseDefaults) {
57358|           // normally GetCacheSizes would return back
    | the actual cache file size
57359|           // as opposed to the default of 20%.
    | However, during rebuild, DevExt->PSMed
57360|           // is not set yet (because we are setting
    | up for it), so in that case we
57361|           // will query the cache file size directly.
57362| Defaults:
57363|           // Params are in MB, convert to number of
    | granules
57364|           DevExt->Cache.PSManBitMapSize =
    | ISC*((1024*1024)/GRANULE_SIZE);
57365|           DevExt->Cache.PSManBitMapMaxSize =
    | MSC*((1024*1024)/GRANULE_SIZE);
57366|       } else {
57367|           LARGE_INTEGER L;
57368|
57369|           // okay use whatever the cache file is at
57370|           ASSERT(DevExt->Cache.CacheFile.Direct);
57371|           Status =
    | DevExt->Cache.CacheFile.Direct->getFileSizeInBytes(L);
57372|           if(NT_SUCCESS(Status)) {
57373|               ASSERT(L.QuadPart);
    | // make sure not zero
57374|               ASSERT((L.QuadPart % (1024*1024)) ==
    | 0); // make sure it is an integer number of
    | megabytes
57375|               ASSERT((L.QuadPart % (GRANULE_SIZE)) ==
    | 0); // make sure it is an integer number of granules
57376|
57377|               DevExt->Cache.PSManBitMapMaxSize =

```

```

    | DevExt->Cache.PSManBitMapSize = (ULONG)(L.QuadPart /
    | GRANULE_SIZE);
57378|     } else {
57379|         ASSERT(FALSE);
57380|         goto Defaults;
57381|     }
57382| }
57383|
57384| #if 1
57385|     //----- 10 November 2001
57386|     // Figure out memory allocation requirements
    | based on worst case
57387|     // granule usage on this volume. In other
    | words, assume that
57388|     // the number of tTreeLeafs needed will never
    | be more than
57389|     // the number of granules in the diff file.
57390|     //
57391|     // However, there is a twist: we also use
    | tTreeLeafs in Direct I/O
57392|     // code. Its usage should be small, unless the
    | volume is extremely
57393|     // fragmented. But this is simply a matter of
    | how fragmented the volume
57394|     // is when PSM is installed. So we add a small
    | safety cushion of 64k of RAM.
57395|
57396|     const ULONG NumMegsNeededForVolume =
    | ConvertNodesToMegs (DevExt->Cache.PSManBitMapSize);
57397|     Status = IncreaseNodeMemory
    | (NumMegsNeededForVolume, 0);
57398|     if ( !NT_SUCCESS(Status) ) {
57399|
    | Debug(DEBUG_DICT,("pd::AllocMemoryForVolume: Error %08x
    | returned from IncreaseNodePoolSize\n",Status));
57400|         try_return (NOTHING);
57401|     }
57402|     //----- 10 November 2001 ----- code
    | change ends
57403| #endif
57404|
57405|     DevExt->Cache.CurrentCacheFileSize = 0;
57406|     DevExt->Cache.PSManBitMapBuffer = NULL;
57407|     DevExt->Cache.PSManCacheCanGrow = FALSE;
57408|
57409|     Debug(DEBUG_DICT,
57410|         ("Cache file size = %d (%d)-%d (%d),
    | alloc mem=%d, threshold=%d,%d\n",
57411|         DevExt->Cache.InitialSize,
57412|         DevExt->Cache.PSManBitMapSize,

```

```

57413|         DevExt->Cache.MaxSize,
57414|         DevExt->Cache.PSManBitMapMaxSize,
57415|         (DevExt->Cache.PSManBitMapMaxSize+1) /
    | NUMBER_OF_BITS_IN_A_BYTE,
57416|         (ULONG)(((unsigned
    | __int64)DevExt->Cache.PSManBitMapSize *
    | DevExt->Cache.CacheWarningThresholdPercent) / 100),
57417|         (ULONG)(((unsigned
    | __int64)DevExt->Cache.PSManBitMapSize *
    | DevExt->Cache.CacheFullThresholdPercent) / 100)
57418|         ));
57419|
57420|         // this can be Paged as only the Thread (which
    | runs at PASSIVE_LEVEL)
57421|         // will access it.
57422|         DevExt->Cache.PSManBitMapBuffer = (PRTL_BITMAP)
    | MemAllocatePoolWithTag( PagedPool,
    | sizeof(RTL_BITMAP)+(((DevExt->Cache.PSManBitMapMaxSize+3
    | 1)/32) *4), BITMAPTAG);
57423|
57424|         if ( DevExt->Cache.PSManBitMapBuffer != NULL )
    | {
57425|         | RtlInitializeBitMap(DevExt->Cache.PSManBitMapBuffer,(PUL
    | ONG)((char*)(DevExt->Cache.PSManBitMapBuffer)+sizeof(RTL
    | _BITMAP)),DevExt->Cache.PSManBitMapMaxSize);
57426|         | RtlClearAllBits(DevExt->Cache.PSManBitMapBuffer);
57427|         DevExt->Cache.CurrentCacheFileSize = 0;
57428|         DevExt->Cache.PSManBitHint = 0;
57429|
57430|         try_return(Status = STATUS_SUCCESS);
57431|     } else {
57432|         Debug(DEBUG_DICT,("Error! Out of paged
    | memory for bitmap\n"));
57433|
57434| #if 1
57435|         // 10 November 2001
57436|         ReclaimNodeMemory (NumMegsNeededForVolume);
    | // clean up node memory that was allocated above.
57437| #endif
57438|
57439|         Status = STATUS_INSUFFICIENT_RESOURCES;
57440|     }
57441|     try_exit:NOTHING;
57442| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
57443|     Status = GetExceptionCode();
57444|     Debug(DEBUG_DICT,("Exception %08x in
    | pd::AllocMemoryForVolume\n",Status));

```

```

57445| }
57446|
57447| Debug(DEBUG_DICT,("pd::AllocMemoryForVolume
    | returning %08x\n",Status));
57448| return Status;
57449| }
57450|
57451| //-----
    | -----
57452|
57453| NTSTATUS PersistentDictionary::FreeMemoryForVolume(
    | PDEVICE_OBJECT Volume )
57454| {
57455|     NTSTATUS Status=STATUS_SUCCESS;
57456|     PFILTERED_EXTENSION DevExt =
        | GetFilteredExtension(Volume);
57457|
57458|     if ( DevExt->Cache.PSManBitMapBuffer ) {
57459|         #if 1
57460|             // 10 November 2001
57461|             const ULONG NumMegsNeededForVolume =
                | ConvertNodesToMegs (DevExt->Cache.PSManBitMapSize);
57462|             ReclaimNodeMemory (NumMegsNeededForVolume);
57463|         #endif
57464|
57465|         MemFreePool(DevExt->Cache.PSManBitMapBuffer);
57466|         DevExt->Cache.PSManBitMapBuffer = NULL;
57467|     }
57468|
57469|     return Status;
57470| }
57471|
57472| //-----
    | -----
57473| // reload stuff from registry
57474|
57475| NTSTATUS PersistentDictionary::GetStateForVolume(
    | PDEVICE_OBJECT Volume )
57476| {
57477|     NTSTATUS Status=STATUS_SUCCESS;
57478|     PFILTERED_EXTENSION DevExt =
        | GetFilteredExtension(Volume);
57479|     WCHAR Buffer[255]={0};
57480|     WCHAR DefaultBuffer[255]={0};
57481|     UNICODE_STRING Reg={0};
57482|     UNICODE_STRING DefaultReg={0};
57483|
57484|     Debug(DEBUG_DICT,("pd::GetStateForVolume:
        | Volume=%08x\n",Volume));
57485|

```



```

57486|   PAGED_CODE();
57487|
57488|   // get system start options
57489|   ASSERT(gRegistryPath.Length<sizeof(Buffer));
57490|
57491|   | RtlCopyMemory(Buffer,gRegistryPath.Buffer,gRegistryPath.
57492|   | Length);
57493|   // NULL terminate, since counted unicode strings
57494|   | are not necessarily null
57495|   // terminated
57496|   Buffer[gRegistryPath.Length / 2] = 0;
57497|   wscat(Buffer,L"\\");
57498|   wcscpy(DefaultBuffer,Buffer);
57499|   wscat(Buffer,DevExt->VolumeGuid);
57500|   wscat(DefaultBuffer,L"persistent");
57501|
57502|   RtlInitUnicodeString(&Reg,Buffer);
57503|   RtlInitUnicodeString(&DefaultReg,DefaultBuffer);
57504|
57505|   // Get defaults first...
57506|
57507|   ULONG D_InitialSize = (ULONG)-20;
57508|   ULONG D_MaxSize = (ULONG)-20;
57509|   ULONG D_CacheWarningThreshold = 80;
57510|   ULONG D_CacheFullThresholdPercent = 90;
57511|   ULONG D_CacheWarningInterval = 15;
57512|   ULONG D_CacheFullActionPercent = 99;
57513|
57514|   BOOLEAN DefaultKeyExists = FALSE;
57515|
57516|   if (
57517|   | RtlCheckRegistryKey(RTL_REGISTRY_ABSOLUTE,DefaultBuffer)
57518|   | ==STATUS_SUCCESS ) {
57519|       DefaultKeyExists = TRUE;
57520|       | Reg_GetULONGKey(&DefaultReg,L"d_InitialSize",D_InitialSi
57521|       | ze,&D_InitialSize);
57522|       | Reg_GetULONGKey(&DefaultReg,L"d_MaxSize",D_MaxSize,&D_Ma
57523|       | xSize);
57524|       | Reg_GetULONGKey(&DefaultReg,L"d_CacheWarningThreshold",D
57525|       | _CacheWarningThreshold,&D_CacheWarningThreshold);
57526|       | Reg_GetULONGKey(&DefaultReg,L"d_CacheWarningInterval",D_
57527|       | CacheWarningInterval,&D_CacheWarningInterval);
57528|

```

```

    | Reg_GetULONGKey(&DefaultReg,L"d_CacheFullActionPercent",
    | D_CacheFullActionPercent,&D_CacheFullActionPercent);
57523|    }
57524|
57525|    // Now override defaults if they exist in the
    | per-volume registry key...
57526|
57527|    if (
    | RtlCheckRegistryKey(RTL_REGISTRY_ABSOLUTE,Buffer)==STATU
    | S_SUCCESS ) {
57528|        UNICODE_STRING Uni;
57529|        WCHAR Temp[255];
57530|
57531|        // per-volume key exists!
57532|
    | Reg_GetULONGKey(&Reg,L"InitialSize",D_InitialSize,&DevEx
    | t->Cache.InitialSize);
57533|
    | Reg_GetULONGKey(&Reg,L"MaxSize",D_MaxSize,&DevExt->Cache
    | .MaxSize);
57534|
    | Reg_GetULONGKey(&Reg,L"CacheWarningThreshold",D_CacheWar
    | ningThreshold,&DevExt->Cache.CacheWarningThresholdPercen
    | t);
57535|
    | Reg_GetULONGKey(&Reg,L"CacheFullThreshold",D_CacheFullTh
    | resholdPercent,&DevExt->Cache.CacheFullThresholdPercent)
    | ;
57536|
    | Reg_GetULONGKey(&Reg,L"CacheWarningInterval",D_CacheWarn
    | ingInterval,&DevExt->Cache.CacheWarningInterval);
57537|
    | Reg_GetULONGKey(&Reg,L"CacheFullAction",CACHE_ACTION_DEN
    | Y_WRITES,&DevExt->Cache.CacheFullAction);
57538|
    | Reg_GetULONGKey(&Reg,L"CacheFullActionPercent",D_CacheFu
    | llActionPercent,&DevExt->Cache.CacheFullActionPercent);
57539|
    | Reg_GetULONGKey(&Reg,L"FileSystem",0,&DevExt->FileSystem
    | );
57540|
57541|    wcscpy(Temp,L"\\?\\Volume{");
57542|    wcscat(Temp,DevExt->VolumeGuid);
57543|    wcscat(Temp,L"}\\");
57544|
57545|    Reg_GetStringKey (
    | &gRegistryPath,L"CacheLocation",Temp,&Uni);
57546|
    | wcscpy(DevExt->Cache.CacheFile.Location,Uni.Buffer);
57547|    Reg_FreeString(&Uni);

```

```

57548|     Reg_GetStringKey (
| &gRegistryPath,L"IndexLocation",Temp,&Uni);
57549|
| wcscpy(DevExt->Cache.IndexFile.Location,Uni.Buffer);
57550|     Reg_FreeString(&Uni);
57551|     Reg_GetStringKey (
| &gRegistryPath,L"HeaderLocation",Temp,&Uni);
57552|
| wcscpy(DevExt->Cache.HeaderFile.Location,Uni.Buffer);
57553|     Reg_FreeString(&Uni);
57554|     Status = STATUS_SUCCESS;
57555|
57556| } else {
57557|     // key doesnt exist, lets make it and add the
| defaults
57558|     Status =
| RtlCreateRegistryKey(RTL_REGISTRY_ABSOLUTE,Buffer);
57559|     ASSERT(Status==0);
57560|     if ( NT_SUCCESS(Status) ) {
57561|         DevExt->Cache.InitialSize = D_InitialSize;
57562|         DevExt->Cache.MaxSize = D_MaxSize;
57563|         DevExt->Cache.CacheWarningThresholdPercent
| = D_CacheWarningThreshold;
57564|         DevExt->Cache.CacheFullThresholdPercent =
| D_CacheFullThresholdPercent;
57565|         DevExt->Cache.CacheWarningInterval =
| D_CacheWarningInterval;
57566|         DevExt->Cache.CacheFullAction =
| CACHE_ACTION_DENY_WRITES;
57567|         DevExt->Cache.CacheFullActionPercent=
| D_CacheFullActionPercent;
57568|         DevExt->FileSystem = FILE_SYSTEM_UNKNOWN;
57569|
57570|         | wcscpy(DevExt->Cache.CacheFile.Location,L"\\??\\Volume{"
| );
57571|         | wcscat(DevExt->Cache.CacheFile.Location,DevExt->VolumeGu
| id);
57572|         | wcscat(DevExt->Cache.CacheFile.Location,L"}\\");
57573|         | wcscpy(DevExt->Cache.HeaderFile.Location,DevExt->Cache.C
| acheFile.Location);
57574|         | wcscpy(DevExt->Cache.IndexFile.Location,DevExt->Cache.Ca
| cheFile.Location);
57575|
57576|         | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Ini

```

```

    | tialSize",REG_DWORD,&DevExt->Cache.InitialSize,sizeof(DW
    | ORD));
57577|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Max
    | Size",REG_DWORD,&DevExt->Cache.MaxSize,sizeof(DWORD));
57578|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Cac
    | heWarningThreshold",REG_DWORD,&DevExt->Cache.CacheWarnin
    | gThresholdPercent,sizeof(DWORD));
57579|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Cac
    | heFullThreshold",REG_DWORD,&DevExt->Cache.CacheFullThres
    | holdPercent,sizeof(DWORD));
57580|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Cac
    | heWarningInterval",REG_DWORD,&DevExt->Cache.CacheWarning
    | Interval,sizeof(DWORD));
57581|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Cac
    | heFullAction",REG_DWORD,&DevExt->Cache.CacheFullAction,s
    | izeof(DWORD));
57582|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Cac
    | heFullActionPercent",REG_DWORD,&DevExt->Cache.CacheFullA
    | ctionPercent,sizeof(DWORD));
57583|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Cac
    | heLocation",REG_SZ,DevExt->Cache.CacheFile.Location,NumB
    | ytes(DevExt->Cache.CacheFile.Location)+sizeof(WCHAR));
57584|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Ind
    | exLocation",REG_SZ,DevExt->Cache.CacheFile.Location,NumB
    | ytes(DevExt->Cache.CacheFile.Location)+sizeof(WCHAR));
57585|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,Buffer,L"Hea
    | derLocation",REG_SZ,DevExt->Cache.CacheFile.Location,Num
    | Bytes(DevExt->Cache.CacheFile.Location)+sizeof(WCHAR));
57586|         Status = STATUS_SUCCESS;
57587|     }
57588| }
57589|
57590|     ULONG BytesReturned=sizeof(Buffer);
57591|     // send the ioctl through us, instead of lower
    | driver
57592|     // so it can update the unique ids
57593|
57594|     Sblo_DeviceIoControl(
    | Volume,IOCTL_MOUNTDEV_QUERY_UNIQUE_ID,NULL,0,(char*)Buff
    | er,sizeof(Buffer),&BytesReturned);
57595|

```

```

57596|    return Status;
57597| }
57598|
57599| //-----
| -----
| -----
57600|
57601| void DeletePsmFilesOnVolume ( PFILTERED_EXTENSION
| DevExt )
57602| {
57603|    // FIXFIXFIX we need to delete map from registry
| (or does sfilter handle that now???)
57604|    __try {
57605|        Debug(DEBUG_DICT,("DeletePsmFiles: DevExt =
| %08x\n", DevExt));
57606|        ASSERT(DevExt->ObjectType ==
| OBJECT_FILTEREDDISK);
57607|
57608|        const WCHAR *const FileTable[] = {
57609|            L"header",
| DevExt->Cache.HeaderFile.FileName,
57610|            L"index",
| DevExt->Cache.IndexFile.FileName,
57611|            L"cache",
| DevExt->Cache.CacheFile.FileName,
57612|            NULL,        NULL
57613|        };
57614|
57615|        for ( ULONG i=0; FileTable[i] != NULL; i += 2 )
| {
57616|            const WCHAR * const Description =
| FileTable[i];
57617|            const WCHAR * const FileName =
| FileTable[i+1];
57618|
57619|            ASSERT(FileName != NULL);
57620|            ASSERT(FileName[0] != L'\0');
57621|
57622|            Debug(DEBUG_DICT,(" delete %S file
| '%S'\n",Description,FileName));
57623|            SbDeleteFile(FileName,
| FILE_ATTRIBUTE_NORMAL);
57624|        }
57625|    } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
57626|        NTSTATUS Status = GetExceptionCode();
57627|        Debug(DEBUG_DICT,("!!!! Exception %08x in
| DeletePsmFiles !!!!\n",Status));
57628|    }
57629| }

```

```

57630|
57631| //-----
    | -----
    | -----
57632|
57633| // Search all the snapshots on the volume to acquire
    | the next in chronological order
57634| // (ie the one with the next lowest
    | OriginalDataSequenceNumber) to the one given.
57635| // If a NULL pointer is given as the start point return
    | the oldest (lowest Sequence number).
57636|
57637| // MUST be called with snapshot resource acquired
    | !!!!!
57638|
57639| pkSnapShotEntry GetNextChronologicalSnapshot (
    | PFILTERED_EXTENSION DevExt, pkSnapShotEntry StartFromSS
    | )
57640| {
57641|     // we can start from 0 to get the oldest snapshot
    | still existing since
57642|     // OriginalDataSequence numbers can never be 0.
57643|     ULONG StartFromSequence =
    | StartFromSS?((pPersistentDictionary)(StartFromSS->Dictio
    | nary))->GetSequenceNumber():0;
57644|
57645|     ULONG EarliestNextSequence = 0xffffffff;
57646|     pkSnapShotEntry q=NULL;
57647|
57648|     pkSnapShotEntry
    | p=GetTopSnapShot(&DevExt->SnapShots);
57649|     while ( p ) {
57650|         ULONG ThisSequence =
    | ((pPersistentDictionary)(p->Dictionary))->GetSequenceNum
    | ber();
57651|         if ( (ThisSequence > StartFromSequence ) &&
    | (ThisSequence < EarliestNextSequence ) ) {
57652|             EarliestNextSequence = ThisSequence;
57653|             UseSnapShot(p);
57654|             if (q) {
57655|                 DoneWithSnapShot(q);
57656|             }
57657|             q=p;
57658|         }
57659|         p=GetNextSnapShot(&DevExt->SnapShots,p);
57660|     }
57661|
57662|     if (StartFromSS) {
57663|         DoneWithSnapShot(StartFromSS);
57664|     }

```

```

57665|
57666|     return q;
57667| }
57668|
57669| //-----
    | -----
    | -----
57670|
57671| void PersistentDictionary::Part2OfRebuildForVolume (
    | PVOID DevObj )
57672| {
57673|
57674|     NTSTATUS Status = 0;
57675|     PDEVICE_OBJECT Volume=(PDEVICE_OBJECT)DevObj;
57676|     PFILTERED_EXTENSION DevExt =
        | GetFilteredExtension(Volume);
57677|     PDEVICE_OBJECT DO = 0;
57678|     ULONG SavedFlags = PSMAN_PSM_FLAGS;
57679|
57680|     if(DevExt->PSMed) {
57681|         Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
            | Volume=%08x '%S'\n",Volume,DevExt->Name));
57682|     } else {
57683|         Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
            | Volume=%08x '%S' not PSMed,
            | skipping\n",Volume,DevExt->Name));
57684|         goto Exit;
57685|     }
57686|
57687|     if(DevExt->IsPhysical) {
57688|         Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
            | Volume=%08x '%S' is physical,
            | skipping\n",Volume,DevExt->Name));
57689|         goto Exit;
57690|     } else {
57691|         Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
            | Volume=%08x '%S' is not
            | physical\n",Volume,DevExt->Name));
57692|     }
57693|
57694|     DO=IoGetAttachedDeviceReference(Volume);
57695|     if(DO) {
57696|         if(DO->Vpb) {
57697|             if(DO->Vpb->Flags & VPB_MOUNTED) {
57698|
            | Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
            | %08x '%S' is vpb mounted\n",Volume,DO,DevExt->Name));
57699|         } else {
57700|
            | Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x

```

```

    | %08x '%S' is not vpb
    | mounted\n",Volume,DO,DevExt->Name));
57701|     }
57702|   } else {
57703|     Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
    | Volume=%08x %08x '%S' has no
    | vpb\n",Volume,DO,DevExt->Name));
57704|   }
57705|   ObDereferenceObject(DO);
57706| } else {
57707|   Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
    | Volume=%08x '%S' Unable to find attached
    | device\n",Volume,DevExt->Name));
57708| }
57709|
57710| #if 1
57711|   if(IsClusterServer()) {
57712|     LARGE_INTEGER TimeToWait;
57713|     ULONG Seconds=5*60;
57714|     // wait 5 minutes, so we can get all the disks
    | online before we map in the snapshots
57715|     | Reg_GetULONGKey(&gRegistryPath,L"MappingDelay",Seconds,&
    | Seconds);
57716|     Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
    | Volume=%08x '%S' waiting %d seconds for all drives to
    | come online\n",Volume,DevExt->Name,Seconds));
57717|     TimeToWait.QuadPart =
    | RELATIVE(SECONDS(Seconds));
57718|     KeDelayExecutionThread(
57719|       (KPROCESSOR_MODE)KernelMode, // IN
    | KPROCESSOR_MODE WaitMode,
57720|       FALSE, // IN BOOLEAN Alertable,
57721|       &TimeToWait ); // IN PLARGE_INTEGER
    | Interval
57722|
57723|     if(!DevExt->PSMed) {
57724|       Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
    | Cluster: Volume=%08x '%S' not PSMed now,
    | skipping\n",Volume,DevExt->Name));
57725|       goto Exit;
57726|     }
57727|   }
57728| #endif
57729|
57730|   if (
    | AcquireOpenCloseResourceOnly(NULL)!=STATUS_WAIT_0 ) {
57731|     goto Exit;
57732|   }
57733|

```



```

57734|   if(DevExt->Dismounting) {
57735|       Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
| Volume=%08x '%S' dismount
| called\n",Volume,DevExt->Name));
57736|       ReleaseOpenCloseResource();
57737|       goto Exit;
57738|   }
57739|
57740|   // Turn off throttling so mapping in occurs faster.
57741|   // This is mostly for cluster to lower the windows
57742|   // of failure until we can eliminante it.
57743|   PSMAN_PSM_FLAGS &= ~PSM_FLAG_PAUSE_ON_IO;
57744|
57745| #ifdef DO_TIMING_TEST
57746|   /*lint -save -e740 */
57747|
| LogError((PDEVICE_OBJECT)PSMAN_DRIVER_OBJECT,NULL,PSM_MAPP
| ING_IN_SNAPSHOTS,0,NULL,0,NULL,0);
57748|   /*lint -restore */
57749| #endif
57750|
57751|   __try {
57752|       START_STOPWATCH(Part2);
57753|       DevExt->OpenCloseAcquired = TRUE;
57754|       UpdateGlobalStatus(PSM_MAPPING_IN_SNAPSHOTS);
57755|
57756|
| if(IsValidHandle(DevExt->Cache.CacheFile.FileHandle)) {
57757|       Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
| Volume=%08x '%S' Part 2 already ran,
| skipping\n",Volume,DevExt->Name));
57758|       try_return(NOTHING);
57759|   }
57760|
57761|   if(GlobalData->ShutDownCalled) {
57762|       Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
| Volume=%08x '%S' shutdown called,
| skipping\n",Volume,DevExt->Name));
57763|       try_return (NOTHING);
57764|   }
57765|
57766|   if(!DevExt->IsMounted) {
57767|       Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
| Volume=%08x '%S' is not mounted,
| skipping\n",Volume,DevExt->Name));
57768|       try_return (NOTHING);
57769|   } else {
57770|       Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
| Volume=%08x '%S' is mounted\n",Volume,DevExt->Name));
57771|   }

```

```

57772|
57773|     ASSERT(DevExt->IsMounted);
57774|
57775|     {
57776|         // get the latest unique id
57777|         CHAR Buffer[256];
57778|         ULONG BytesReturned=sizeof(Buffer);
57779|         Sblo_DeviceIoControl(
57780|             | Volume,IOCTL_MOUNTDEV_QUERY_UNIQUE_ID,NULL,0,(char*)Buff
57781|             | er,sizeof(Buffer),&BytesReturned);
57782|     }
57783|     Status = OpenFilesForVolume(Volume);
57784|     if(NT_SUCCESS(Status)) {
57785|         | InsertFileObjectToSkip(DevExt->Cache.HeaderFile.FileObje
57786|         | ct);
57787|         | InsertFileObjectToSkip(DevExt->Cache.IndexFile.FileObjec
57788|         | t);
57789|         | InsertFileObjectToSkip(DevExt->Cache.CacheFile.FileObjec
57790|         | t);
57791|         | Debug(DEBUG_DEVCON,("Part2OfRebuildForVolume:
57792|         | directio=%d, Setting to false\n",DevExt->DoDirectIO));
57793|         DevExt->DoDirectIO=FALSE;
57794|         // Add snapshot volumes without virtual
57795|         | writes for freespace logic
57796|         // Dismount snapshot volumes
57797|         // Mount snapshot volumes with virtual
57798|         | writes
57799|         // now that all the nodes have been read
57800|         | into memory and the snapshot images will be correct,
57801|         | lets
57802|         // add the virtual drives to the system
57803|         Debug(DEBUG_DICT,("Adding virtual drives to
57804|         | system\n"));
57805|         if ( DoFreeSpaceChecks() ) {
57806|             gVDiskDoVirtualIO = FALSE;
57807|         }
57808|         START_STOPWATCH(Part2_AddingStage1);
57809|         GetSnapShotForWrite();
57810|         __try {
57811|             PDEVICE_OBJECT Virtual;
57812|             PRTL_BITMAP CachingMapInRebuild = NULL;

```

```

57807|          PRTL_BITMAP MapToBeReplaced = NULL;
57808|          pPersistentDictionary Dict = NULL;
57809|
57810|          // we can start from 0 to get the
| oldest snapshot still existing since
57811|          // OriginalDatasequence numbers can
| never be 0.
57812|          pkSnapShotEntry
| p=GetNextChronologicalSnapshot(DevExt, NULL);
57813|          while ( p ) {
57814|
57815|              Dict =
| ((pPersistentDictionary)(p->Dictionary));
57816|
57817| #if 0 // don't need this code now as we're rebuilding
| our map in private till it's finished
57818|
57819|          // Going into map update switch
| maps to what it expects we would have already done at
| the moment we froze the snapshot.
57820|          Dict->Shared->MapInTransform =
| Dict->Shared->Map;
57821|          Dict->Shared->Map = NULL;
57822| #endif
57823|
57824|          START_STOPWATCH(Part2_AddDrive);
57825|          NTSTATUS Status = TdAddDrive (
| &CachingMapInRebuild, p->DeviceObject,
| p->MasterSnapShot, &Virtual,TRUE);
57826|          PAUSE_STOPWATCH(Part2_AddDrive);
57827|
57828|          if ( NT_SUCCESS(Status) ) {
57829|              Debug(DEBUG_DICT,("Success
| adding virtual drive %08x to system, seq=%08x
| (%d)\n",Virtual,
57830|
| ((pPersistentDictionary)(p->Dictionary))->GetSequenceNum
| ber(),p->MasterSnapShot->Instance));
57831|
57832|          } else {
57833|              Debug(DEBUG_DICT,("Error %08x
| adding virtual drive to system, seq=%08x
| (%d)\n",Status,
57834|
| ((pPersistentDictionary)(p->Dictionary))->GetSequenceNum
| ber(),p->MasterSnapShot->Instance));
57835|          }
57836|
57837|          ULONG SSJustAdded =
| Dict->GetSequenceNumber();

```

```

57838|
57839|         p =
57840|         | GetNextChronologicalSnapshot(DevExt, p);
57841|         if ( DoFreeSpaceChecks() ) {
57842|             START_STOPWATCH(Part2_Credit);
57843|             MyAcquireResourceExclusiveLite
57844|             | ( &Dict->Shared->TreeResource, TRUE );
57845|             __try {
57846|                 | CreditInterveningGranulesInTree (CachingMapInRebuild,
57847|                 | &Dict->Shared->Tree ,SSJustAdded,
57848|                 | p!=NULL?((pPersistentDictionary)(p->Dictionary))->GetSeq
57849|                 | uenceNumber():0 );
57850|                 } __finally {
57851|                 | MyReleaseResourceForThreadLite (
57852|                 | &Dict->Shared->TreeResource );
57853|                 }
57854|                 PAUSE_STOPWATCH(Part2_Credit);
57855|             }
57856|             if(GlobalData->ShutDownCalled) {
57857|                 | Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
57858|                 | '%S' shutdown called during add
57859|                 | drive\n",Volume,DevExt->Name));
57860|                 if ( p ) {
57861|                     DoneWithSnapShot(p);
57862|                 }
57863|                 try_return (NOTHING);
57864|             }
57865|             if(DevExt->Dismounting) {
57866|                 | Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
57867|                 | '%S' dismount called\n",Volume,DevExt->Name));
57868|                 if(p) {
57869|                     DoneWithSnapShot(p);
57870|                 }
57871|                 try_return (NOTHING);
57872|             }
57873|             } // while
57874|             // if we're here we must have completed
57875|             | building a new map
57876|             ASSERT (!p && CachingMapInRebuild &&
57877|             | Dict);
57878|             // so set it in service
57879|             MapToBeReplaced =

```

```

    | Dict->GetVolumeCachingMap( 0 );
57873|         Dict->SetVolumeCachingMap( 0,
    | CachingMapInRebuild );
57874|         if (MapToBeReplaced != NULL) {
57875|             Debug(DEBUG_DICT,("Freeing obsolete
    | map at 0x%08x . . \n", MapToBeReplaced ));
57876|             MemFreePool(MapToBeReplaced);
57877|         }
57878|         Debug(DEBUG_DICT,("Done adding virtual
    | drives to system\n"));
57879|
57880|     } __finally {
57881|         PAUSE_STOPWATCH(Part2_AddingStage1);
57882|         ReleaseSnapShotForWrite();
57883|         gVDiskDoVirtualIO=TRUE;
57884|     }
57885|
57886|
57887|
57888|         // okay, now that we have our bitmaps, lets
    | go through and dismount/remount all the
57889|         // volumes using virtual writes this time.
57890|         if ( DoFreeSpaceChecks() ) {
57891|
57892|             START_STOPWATCH(Part2_AddingStage2);
57893|             GetSnapShotForRead();
57894|             __try {
57895|                 START_STOPWATCH(Part2_DismountAll);
57896|                 gVDiskDoVirtualIO=FALSE;
57897|                 __try {
57898|
    | Rebuild_DismountAllVolumes(Volume,TRUE);
57899|                 } __finally {
57900|
    | PAUSE_STOPWATCH(Part2_DismountAll);
57901|                 gVDiskDoVirtualIO=TRUE;
57902|                 if(AbnormalTermination()) {
57903|
    | Rebuild_ReenableVolumeMounts(Volume);
57904|                 }
57905|                 }
57906|
57907|
    | Rebuild_ReenableVolumeMounts(Volume);
57908|
57909|         if(GlobalData->ShutDownCalled) {
57910|
    | Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
    | '%S' shutdown called during remount for
    | freespace\n",Volume,DevExt->Name));

```

```

57911|         try_return (NOTHING);
57912|     }
57913|
57914|         if(DevExt->Dismounting) {
57915|
57916|         | Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
57917|         | '%S' dismount called\n",Volume,DevExt->Name));
57918|         try_return (NOTHING);
57919|     }
57920|
57921|     START_STOPWATCH(Part2_MountAll);
57922|     EnableWritesToNewFiles();
57923|     __try {
57924|         | Rebuild_MountAllVolumes(Volume);
57925|     } __finally {
57926|         DisableWritesToNewFiles();
57927|     }
57928|     PAUSE_STOPWATCH(Part2_MountAll);
57929| } __finally {
57930|
57931|     | PAUSE_STOPWATCH(Part2_AddingStage2);
57932|     ReleaseSnapShotForRead();
57933| }
57934|     Debug(DEBUG_DICT,("Done adding real
57935|     | virtual drives to system\n"));
57936| }
57937|
57938|     if(GlobalData->ShutDownCalled) {
57939|
57940|         | Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
57941|         | '%S' shutdown called during recreate of
57942|         | junctions\n",Volume,DevExt->Name));
57943|         try_return (NOTHING);
57944|     }
57945|
57946|         if(DevExt->Dismounting) {
57947|
57948|         | Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
57949|         | '%S' dismount called\n",Volume,DevExt->Name));
57950|         try_return (NOTHING);
57951|     }
57952|
57953|         Debug(DEBUG_DICT,("Recreating junction
57954|         | points\n"));
57955|         START_STOPWATCH(Part2_RecreateJunctions);
57956|         GetSnapShotForRead();
57957|         __try {
57958|             // recreate junctions incase they
57959|             | changed

```

```

57949|          // which can happen on cluster server
57950|          Rebuild_RecreateJunctionsOnVolume(
57951|      | Volume );
57952|      } __finally {
57953|          PAUSE_STOPWATCH(Part2_RecreateJunctions);
57954|          ReleaseSnapShotForRead();
57955|      }
57956|      Debug(DEBUG_DICT,("Done Recreating junction
57957|      | points\n"));
57958|      if(GlobalData->ShutDownCalled) {
57959|          Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
57960|          | '%S' shutdown called during Notify user of
57961|          | event\n",Volume,DevExt->Name));
57962|          try_return (NOTHING);
57963|      }
57964|      if(DevExt->Dismounting) {
57965|          Debug(DEBUG_DICT,("Part2OfRebuildForVolume: Volume=%08x
57966|          | '%S' dismount called\n",Volume,DevExt->Name));
57967|          try_return (NOTHING);
57968|      }
57969|      NotifyUserModeOfVolumeOnlineEvent(DevExt);
57970|      } else {
57971|          Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
57972|          | error %08x opening files\n",Status));
57973|      }
57974|      Debug(DEBUG_DICT,("Part2OfRebuildForVolume:
57975|      | Volume=%08x '%S' leaving\n",Volume,DevExt->Name));
57976|      try_exit: NOTHING;
57977|
57978|      PAUSE_STOPWATCH(Part2);
57979|      STOPWATCH_DUMPALL();
57980|      STOPWATCH_RESETALL();
57981|
57982|      } __except(
57983|          | ExceptionFilter(GetExceptionInformation()) ) {
57984|          Debug(DEBUG_SFILTER,("Exception %08x in
57985|          | Part2OfRebuildForVolume\n",GetExceptionCode()));
57986|      }
57987|
57988|      #ifdef DO_TIMING_TEST
57989|      /*lint -save -e740 */
57990|
57991|      | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_DONE
57992|      | _MAPPING_IN_SNAPSHOTS,0,NULL,0,NULL,0);

```

```

57985|  /*lint -restore */
57986| #endif
57987|
57988|  // restore the global flags
57989|  PSMAN_PSM_FLAGS = SavedFlags;
57990|  DevExt->OpenCloseAcquired = FALSE;
57991|  ReleaseOpenCloseResource();
57992|
57993|  // GlobalStatus needs to be updated at bootup time.
57994|  UpdateGlobalStatus(PSM_IDLE);
57995|
57996| Exit:
57997|
57998|  PsTerminateSystemThread( 0 );
57999| }
58000|
58001| void PersistentDictionary::SetSystemReady(void)
58002| {
58003|  Debug(DEBUG_DICT,("!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    | pd::SetSystemReady: System Is Ready
    | !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!\n"));
58004|  SystemReady = TRUE;
58005| }
58006|
58007| NTSTATUS RemoveVolumesFromSystem( PDEVICE_OBJECT
    | Volume);
58008|
58009| NTSTATUS
    | PersistentDictionary::CopyClusterRegistryToLocalRegistry
    | ( PDEVICE_OBJECT Volume, PUNICODE_STRING LocalReg )
58010| {
58011|  PFILTERED_EXTENSION DevExt =
    | GetFilteredExtension(Volume);
58012|  KEY_VALUE_FULL_INFORMATION *Handle;
58013|  WCHAR Buffer[256];
58014|  UNICODE_STRING ClusterReg;
58015|  RETRIEVAL_POINTERS_BUFFER *RP = NULL;
58016|
58017|  ASSERT(DevExt->VolumeGuid[0]);
58018|  ASSERT(DevExt->UniqueId[0]);
58019|
58020|
    | wcscpy(Buffer,L"\\Registry\\Machine\\Cluster\\Persistent
    | StorageManager\\");
58021|  wcscat(Buffer,DevExt->UniqueId);
58022|
58023|  RtlInitUnicodeString(&ClusterReg,Buffer);
58024|
58025|  // create local registry
58026|  GetStateForVolume(Volume);

```



```

58027|
58028|     if (
        | NT_SUCCESS(Reg_GetBinaryKey(&ClusterReg,L"HeaderMap",(PV
        | OID*)&RP,(PVOID*)&Handle)) ) {
58029|
        | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
        | ffer,L"HeaderMap",REG_BINARY,RP,Handle->DataLength);
58030|         Reg_FreeBinary(Handle);
58031|     } else {
58032|
        | Debug(DEBUG_DICT,("pd::CopyClusterRegistryToLocalRegistr
        | y: Deleting Local HeaderMap\n"));
58033|
        | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->B
        | uffer,L"HeaderMap");
58034|     }
58035|     if (
        | NT_SUCCESS(Reg_GetBinaryKey(&ClusterReg,L"IndexMap",(PVO
        | ID*)&RP,(PVOID*)&Handle)) ) {
58036|
        | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
        | ffer,L"IndexMap",REG_BINARY,RP,Handle->DataLength);
58037|         Reg_FreeBinary(Handle);
58038|     } else {
58039|
        | Debug(DEBUG_DICT,("pd::CopyClusterRegistryToLocalRegistr
        | y: Deleting Local IndexMap\n"));
58040|
        | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->B
        | uffer,L"IndexMap");
58041|     }
58042|     if (
        | NT_SUCCESS(Reg_GetBinaryKey(&ClusterReg,L"CacheMap",(PVO
        | ID*)&RP,(PVOID*)&Handle)) ) {
58043|
        | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
        | ffer,L"CacheMap",REG_BINARY,RP,Handle->DataLength);
58044|         Reg_FreeBinary(Handle);
58045|     } else {
58046|
        | Debug(DEBUG_DICT,("pd::CopyClusterRegistryToLocalRegistr
        | y: Deleting Local CacheMap\n"));
58047|
        | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->B
        | uffer,L"CacheMap");
58048|     }
58049|
58050|     ULONG Value;
58051|
58052|

```

```

    | Reg_GetULONGKey(&ClusterReg,L"InitialSize",DevExt->Cache
    | .InitialSize, &Value);
58053|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"InitialSize",REG_DWORD,&Value,sizeof(DWORD));
58054|
58055|
    | Reg_GetULONGKey(&ClusterReg,L"MaxSize",DevExt->Cache.Max
    | Size,&Value);
58056|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"MaxSize",REG_DWORD,&Value,sizeof(DWORD));
58057|
58058|
    | Reg_GetULONGKey(&ClusterReg,L"CacheWarningThreshold",Dev
    | Ext->Cache.CacheWarningThresholdPercent,&Value);
58059|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"CacheWarningThreshold",REG_DWORD,&Value,sizeof(DW
    | ORD));
58060|
58061|
    | Reg_GetULONGKey(&ClusterReg,L"CacheFullThreshold",DevExt
    | ->Cache.CacheFullThresholdPercent,&Value);
58062|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"CacheFullThreshold",REG_DWORD,&Value,sizeof(DWORD
    | ));
58063|
58064|
    | Reg_GetULONGKey(&ClusterReg,L"CacheWarningInterval",DevE
    | xt->Cache.CacheWarningInterval,&Value);
58065|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"CacheWarningInterval",REG_DWORD,&Value,sizeof(DWO
    | RD));
58066|
58067|
    | Reg_GetULONGKey(&ClusterReg,L"CacheFullAction",DevExt->C
    | ache.CacheFullAction,&Value);
58068|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"CacheFullAction",REG_DWORD,&Value,sizeof(DWORD));
58069|
58070|
    | Reg_GetULONGKey(&ClusterReg,L"CacheFullActionPercent",De
    | vExt->Cache.CacheFullActionPercent,&Value);
58071|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"CacheFullActionPercent",REG_DWORD,&Value,sizeof(D

```

```

    | WORD));
58072|
58073|
    | Reg_GetULONGKey(&ClusterReg,L"FileSystem",DevExt->FileSy
    | stem, &Value);
58074|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"FileSystem",REG_DWORD,&Value,sizeof(DWORD));
58075|
58076| #if 0
58077|     UNICODE_STRING Uni;
58078|     // dont do this as the registry entries are
    | specific to each node, not the volume
58079|     // this means that on cluster machines, the cache
    | file must be on the volume
58080|     // that has the snapshots.
58081|     Reg_GetStringKey (
    | &ClusterReg,L"HeaderLocation",DevExt->Cache.HeaderFile.L
    | ocation,&Uni);
58082|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"HeaderLocation",REG_SZ,Uni.Buffer,NumBytes(Uni.Bu
    | ffer)+sizeof(WCHAR));
58083|     Reg_FreeString(&Uni);
58084|
58085|     Reg_GetStringKey (
    | &ClusterReg,L"IndexLocation",DevExt->Cache.IndexFile.Loc
    | ation,&Uni);
58086|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"IndexLocation",REG_SZ,Uni.Buffer,NumBytes(Uni.Buf
    | fer)+sizeof(WCHAR));
58087|     Reg_FreeString(&Uni);
58088|
58089|     Reg_GetStringKey (
    | &ClusterReg,L"CacheLocation",DevExt->Cache.CacheFile.Loc
    | ation,&Uni);
58090|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg->Bu
    | ffer,L"CacheLocation",REG_SZ,Uni.Buffer,NumBytes(Uni.Buf
    | fer)+sizeof(WCHAR));
58091|     Reg_FreeString(&Uni);
58092| #endif
58093|
58094|     // refresh with new data from cluster database
58095|     GetStateForVolume(Volume);
58096|     return STATUS_SUCCESS;
58097| }
58098|
58099| //-----

```

```

| -----
| -----
58100| // This function will determine if we need to load the
| snapshots
58101| // without touching the actual volume.
58102|
58103| BOOLEAN PersistentDictionary::CheckIfLoadNeeded(
| PDEVICE_OBJECT Volume )
58104| {
58105|     PAGED_CODE();
58106|
58107|     Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded:
| Volume=%08x\n",Volume));
58108|
58109|     if(GlobalData->ShutDownCalled) {
58110|         // dont load during shutdown
58111|         Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded:
| Shutdown - returning FALSE\n"));
58112|         return FALSE;
58113|     }
58114|
58115|     PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
58116|     if(DevExt->VolumeGuid[0]==0) {
58117|         // no volume guid defined
58118|         Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded: No
| volume Guid - returning FALSE\n"));
58119|         return FALSE;
58120|     }
58121|
58122|     WCHAR Buffer[255]={0};
58123|     PVOID Handle;
58124|     RETRIEVAL_POINTERS_BUFFER *RP = NULL;
58125|     UNICODE_STRING Reg={0};
58126|     BOOLEAN ClusterExists=FALSE;
58127|
58128|     // if cluster exists, check its map
58129|
58130|
| wscpy(Buffer,L"\\Registry\\Machine\\Cluster\\Persistent
| StorageManager\\");
58131|     wcscat(Buffer,DevExt->Uniqueid);
58132|
58133|     if(DevExt->Uniqueid[0]!=0) {
58134|         RtlInitUnicodeString(&Reg,Buffer);
58135|         // if cluster entry exists, use it instead, as
| the local copy may be stale
58136|
| if(RtlCheckRegistryKey(RTL_REGISTRY_ABSOLUTE,Buffer)==ST
| ATUS_SUCCESS) {

```

```

58137|         if (
| INT_SUCCESS(Reg_GetBinaryKey(&Reg,L"HeaderMap",(PVOID*)&
| RP,&Handle)) ) {
58138|         | Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded: Cluster
| registry has no maps - returning FALSE\n"));
58139|         // no maps
58140|         return FALSE;
58141|     }
58142|     Reg_FreeBinary(Handle);
58143|     Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded:
| Cluster registry valid\n"));
58144|     ClusterExists=TRUE;
58145| }
58146| }
58147|
58148| ASSERT(gRegistryPath.Length<sizeof(Buffer));
58149|
| RtlCopyMemory(Buffer,gRegistryPath.Buffer,gRegistryPath.
| Length);
58150| Buffer[gRegistryPath.Length / 2] = 0;
58151| wscat(Buffer,L"\");
58152| wscat(Buffer,DevExt->VolumeGuid);
58153|
58154| // check local maps
58155| RtlInitUnicodeString(&Reg,Buffer);
58156|
58157| if (
| RtlCheckRegistryKey(RTL_REGISTRY_ABSOLUTE,Buffer)!=STATU
| S_SUCCESS ) {
58158|     if(ClusterExists) {
58159|         // okay, we have cluster registry but no
| local registry, lets create it
58160|         Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded:
| No local registry for volume, copying cluster down to
| local\n"));
58161|         | CopyClusterRegistryToLocalRegistry(Volume,&Reg);
58162|     } else {
58163|         Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded:
| No local registry for volume - returning FALSE\n"));
58164|         // key doesnt exist
58165|         return FALSE;
58166|     }
58167| } else {
58168|     if(ClusterExists) {
58169|         Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded:
| Updating local with cluster database\n"));
58170|         // update the local based on the cluster
58171|

```

```

    | CopyClusterRegistryToLocalRegistry(Volume,&Reg);
58172|     }
58173| }
58174|
58175| if (
    | NT_SUCCESS(Reg_GetBinaryKey(&Reg,L"HeaderMap",(PVOID*)&R
    | P,&Handle)) ) {
58176|     Reg_FreeBinary(Handle);
58177|     Handle = NULL;
58178| } else {
58179|     // no maps
58180|     Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded:
    | Local registry has no maps - returning FALSE\n"));
58181|     return FALSE;
58182| }
58183|
58184| BOOLEAN LoadNeeded = TRUE;
58185|
58186| #if ENABLE_INDEX_LOAD_WHEEL
58187|     // If this code block is enabled, it means that we
    | check the IndexLoadWheel on disk and in memory
58188|     // to see if they match. If not, we assume the
    | index file is dirty and must be reloaded.
58189|     // Otherwise, the index file is assumed to be in
    | sync with in-memory snapshot data structures,
58190|     // so there is no reason to reload it.
58191|
58192|     if ( DevExt->Cache.Header == NULL ) {
58193|         Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded: Need
    | to load because DevExt=%08x does not yet have a
    | header!\n",DevExt));
58194|     } else {
58195|         // Need to compare dirty counters from on-disk
    | header and in-memory header.
58196|
58197|         pHeader MostRecentValidHeader=NULL,
    | SmallMostRecentValidHeader=NULL;
58198|         NTSTATUS ReadHeaderStatus = ReadHeader (Volume,
    | MostRecentValidHeader, SmallMostRecentValidHeader);
58199|         if ( NT_SUCCESS(ReadHeaderStatus) ) {
58200|             ASSERT(MostRecentValidHeader != NULL);
58201|             ASSERT(SmallMostRecentValidHeader != NULL);
58202|
58203|             if ( MostRecentValidHeader!=NULL &&
    | SmallMostRecentValidHeader!=NULL ) {
58204|
    | Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded: Dirty
    | counters - OnDisk=%08x
    | InMemory=%08x\n",MostRecentValidHeader->IndexLoadWheel,D
    | evExt->Cache.Header->IndexLoadWheel));

```

```

58205|         if (
| MostRecentValidHeader->IndexLoadWheel ==
| DevExt->Cache.Header->IndexLoadWheel) {
58206|         // Getting here means the index
| file on disk matches the rbtrees already in memory.
58207|         | Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded: Index file
| on disk has not changed since last reload.\n"));
58208|         LoadNeeded = FALSE;
58209|         } else {
58210|         | Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded: Index file
| on disk has CHANGED! Need to do index reload.\n"));
58211|         }
58212|     }
58213| } else {
58214|     // we will say to reload, incase snapshots
| are on a disk that has disappeared (as on cluster)
58215|     Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded:
| Header file is unreadable.\n"));
58216|     ASSERT(MostRecentValidHeader == NULL);
58217|     ASSERT(SmallMostRecentValidHeader == NULL);
58218| }
58219|
58220| if ( MostRecentValidHeader != NULL ) {
58221|     FREE_POINTER (MostRecentValidHeader);
58222| }
58223|
58224| if ( SmallMostRecentValidHeader != NULL ) {
58225|     FREE_POINTER (SmallMostRecentValidHeader);
58226| }
58227| }
58228| #endif /*ENABLE_INDEX_LOAD_WHEEL*/
58229|
58230| Debug(DEBUG_DICT,("pd::CheckIfLoadNeeded returning
| %s\n", (LoadNeeded?"TRUE":"FALSE")));
58231| return LoadNeeded;
58232| }
58233|
58234| //-----
| -----
| -----
58235|
58236| #if DEBUG_VALIDATE_DIFF_GRANULES
58237|
58238| bool PersistentDictionary::ValidateDiffGranule (
58239|     tIndexSector      *indexSector,
58240|     void              *diffGranuleBuffer,
58241|     PFILTERED_EXTENSION devExt,
58242|     ULONG              granuleNumber )

```

```

58243| {
58244|     // Read the granule from the diff file.
58245|     bool diffGranuleIsValid = false;    // unless
        | proven correct
58246|
58247|     START_STOPWATCH(Rebuild_ValidateDiffGranules);
58248|
58249|     IO_STATUS_BLOCK IoStatus;
58250|     LARGE_INTEGER Location;
58251|
58252|     Location.QuadPart = __int64(GRANULE_SIZE) *
        | __int64(granuleNumber);
58253|
58254|     NTSTATUS Status = PsmReadFromFile(
58255|         &devExt->Cache.CacheFile,
58256|         &IoStatus,
58257|         diffGranuleBuffer,
58258|         GRANULE_SIZE,
58259|         &Location,
58260|         devExt->DoDirectIO,
58261|         &devExt->Cache.DirectAccessResource );
58262|
58263|     if ( NT_SUCCESS(Status) ) {
58264|         BOOLEAN granuleIsValid = ValidateChecksum (
58265|             GRANULE_SIZE,
58266|             diffGranuleBuffer,
58267|             indexSector->Info.DiskNode[0].DataChecksum
        | );
58268|
58269|         if ( granuleIsValid ) {
58270|             diffGranuleIsValid = true;
58271|             Debug(DEBUG_DICT,("pd::ValidateDiffGranule:
        | Position=%08x, DiskGranule=%08x is valid\n",
        | granuleNumber,
        | indexSector->Info.DiskNode[0].DasdGranule));
58272|         } else {
58273|             Debug(DEBUG_DICT,("pd::ValidateDiffGranule:
        | !!! DATA CORRUPTION !!! Position=%08x,
        | DiskGranule=%08x\n", granuleNumber,
        | indexSector->Info.DiskNode[0].DasdGranule));
58274|             Debug(DEBUG_DICT,("pd::ValidateDiffGranule:
        | Dump of granule data follows:\n"));
58275|             DumpSector ( (char *)diffGranuleBuffer,
        | GRANULE_SIZE );
58276|         }
58277|     } else {
58278|         Debug(DEBUG_DICT,("pd::ValidateDiffGranule:
        | !!! Error %08x reading from diff file!
        | DevExt=%08x\n", Status, devExt));
58279|     }

```



```

58280|
58281|    PAUSE_STOPWATCH(Rebuild_ValidateDiffGranules);
58282|
58283|    return diffGranuleIsValid;
58284| }
58285|
58286| #endif /*DEBUG_VALIDATE_DIFF_GRANULES*/
58287|
58288|
58289| //-----
| -----
| -----
58290| // This function is called in 2 cases:
58291| //
58292| // Rebuild = TRUE : The volume has just been requested
| to be mounted. The volume is not yet mounted so no
58293| //          requests can be sent down that
| would cause a mount or a deadlock will occur
58294| //
58295| // Rebuild = FALSE: A snapshot is being created after
| the volume has been mounted. IOW, a snapshot was just
58296| //          scheduled to be created
58297|
58298| NTSTATUS
| PersistentDictionary::RebuildSnapShotsForVolume(
58299|     PDEVICE_OBJECT    Volume,
58300|     BOOLEAN            Rebuild,
58301|     PVOID              AbortEvent )
58302| {
58303|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
58304|     PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
58305|     ULONG numberOfNodesInserted = 0;
58306|     ULONG numberOfFailedInserts = 0;
58307|     ULONG numberOfVirtualWrites = 0;
58308|
58309|     Debug(DEBUG_DICT,("pd::RebuildSnapShotsForVolume:
| Volume=%08x '%S', Rebuild=%s, rc=%08x\n",
58310|         Volume,DevExt->Name,
58311|         (Rebuild ? "TRUE" : "FALSE"),
58312|         DevExt->Cache.ReferenceCount));
58313|
58314|     START_STOPWATCH(Rebuild_Total);
58315|
58316| #if DEBUG_VALIDATE_DIFF_GRANULES
58317|     void *diffGranuleBuffer =
| MemAllocatePoolWithTag(PagedPool,GRANULE_SIZE,TEMPTAG);
58318|     Debug(DEBUG_DICT,("pd::RebuildSnapShotsForVolume:
| diffGranuleBuffer=%08x\n", diffGranuleBuffer));
58319|     ASSERT (diffGranuleBuffer != NULL);

```

```

58320| #endif /*DEBUG_VALIDATE_DIFF_GRANULES*/
58321|
58322|     __try {
58323|         if(GlobalData->ShutDownCalled) {
58324|
58325|             | Debug(DEBUG_DICT,("pd::RebuildSnapShotsForVolume:
58326|             | shutdown called during
58327|             | rebuild\n",Volume,DevExt->Name));
58328|             Status = STATUS_CANCELLED;
58329|             try_return (NOTHING);
58330|         }
58331|         // if we are remounting and still have stuff
58332|         | hanging around
58333|         // then we were not notified of the dismount
58334|         if(Rebuild && DevExt->Cache.ReferenceCount) {
58335|
58336|             | Debug(DEBUG_DICT,("pd::RebuildSnapShotsForVolume: Told
58337|             | to rebuild with active snapshots, removing them\n"));
58338|             UnloadSnapShotsForVolume(Volume,TRUE);
58339|             ASSERT(DevExt->Cache.ReferenceCount==0);
58340|         }
58341|
58342|         if(DevExt->Cache.ReferenceCount==0) {
58343|
58344|             | UpdateGlobalStatus(PSM_RESOURCE_ACQUISITION);
58345|             Status = GetStateForVolume(Volume);
58346|             if ( NT_SUCCESS(Status) ) {
58347| DoOpen:
58348|                 if(Rebuild) {
58349|                     // We are executing BEFORE the
58350|                     | volume is mounted
58351|                     // so lets do direct io until after
58352|                     | it is mounted
58353|                     Status =
58354|                     | PersistentDictionary::RetrieveDirectIOMaps(Volume);
58355|                     if(NT_SUCCESS(Status)) {
58356|                         Debug(DEBUG_DICT,("Success
58357|                         | getting map, setting directio=true\n"));
58358|                         DevExt->DoDirectIO = TRUE;
58359|                     } else {
58360|                         Debug(DEBUG_DICT,("Error %08x
58361|                         | getting map, leaving
58362|                         | directio=%d\n",Status,DevExt->DoDirectIO));
58363|                     }
58364|                 } else {
58365|                     // cache file handles, bitmap,
58366|                     | etc...
58367|                     if ( QueryResetPsm() ) {
58368|                         Status = STATUS_UNSUCCESSFUL;
58369|                     } else {

```

```

58356|             Status =
| OpenFilesForVolume(Volume);
58357|             if ( NT_SUCCESS(Status) ) {
58358|                 | Debug(DEBUG_DEVCON,("RebuildSnapShotsForVolume:
| directio=%d, Setting to false\n",DevExt->DoDirectIO));
58359|                 DevExt->DoDirectIO = FALSE;
58360|                 // dont care about status
| here.
58361|                 // we also need to have it
| read the local registry
58362|                 // as the cluster registry
| may not have been copied yet if
58363|                 // this is the first
| snapshot on the volume
58364|                 | PersistentDictionary::RetrieveDirectIOMaps(Volume,FALSE)
| ;
58365|                 }
58366|             }
58367|         }
58368|
58369|         if ( NT_SUCCESS(Status) ) {
58370|             if ( (Rebuild) && (QueryNoPsm() ||
| QueryResetPsm()) ) {
58371|                 Status =
| STATUS_OBJECT_NAME_NOT_FOUND;
58372|             }
58373|
58374|             if ( NT_SUCCESS(Status) ) {
58375|                 Status = LoadHeader(Volume);
58376|                 if ( NT_SUCCESS(Status) ) {
58377|                     if(!Rebuild) {
58378|
| InsertFileObjectToSkip(DevExt->Cache.HeaderFile.FileObje
| ct);
58379|
| InsertFileObjectToSkip(DevExt->Cache.IndexFile.FileObjec
| t);
58380|
| InsertFileObjectToSkip(DevExt->Cache.CacheFile.FileObjec
| t);
58381|                 }
58382|
58383|                 // this code was added to
| solve a problem with allocating too
58384|                 // much memory if the
| volume was extended. If we have snapshots
58385|                 // we need to allocate
| memory based on the size of the file and

```

```

58386|                // not what the user
| configured it for (ie 20%).
58387|                // the reason too much
| memory is a problem is that it sets the
58388|                // PsManBitMap[Max]Size to
| the new value, so we think our cache file
58389|                // size is larger than it
| actually is.
58390|                if(Rebuild) {
58391|                    if (
| !IsListEmpty(&DevExt->Cache.SnapShotHead) ) {
58392|                        // snapshots exist
58393|                        Status =
| AllocMemoryForVolume(Volume,FALSE);
58394|                    } else {
58395|                        // no snapshots
| exist
58396|                        Status =
| AllocMemoryForVolume(Volume,TRUE);
58397|                    }
58398|                } else {
58399|                    // not rebuilding so no
| snapshots exist
58400|                    Status =
| AllocMemoryForVolume(Volume,TRUE);
58401|                }
58402|
58403|                if ( NT_SUCCESS(Status) ) {
58404|                    // Shared->BitMap =
| alloc mem
58405|                    // init bitmap, etc..
58406|                    PsmBitMapValidate
| (DevExt->Cache.PSManBitMapBuffer);
58407|                    RtlClearAllBits (
| DevExt->Cache.PSManBitMapBuffer );
58408|
| DevExt->Cache.CurrentCacheFileSize = 0;
58409|
| DevExt->Cache.PSManBitHint = 0;
58410|
58411|                // if rebuilding and
| there are snapshots, load them
58412|                if (Rebuild) {
58413|                    if (
| !IsListEmpty(&DevExt->Cache.SnapShotHead) ) {
58414|                        pOT_USER User =
| FindPSMUser(GlobalSystemProcessId,(_ETHREAD*)-2);
58415|                        ASSERT(User);
58416|                        ULONG
| NumSnapShotsAdded=0;

```

```

58417|
58418|
58419|         | UpdateGlobalStatus(PSM_LOADING_SNAPSHOTS);
58419|         // go through
58419|         | header adding all snapshots into memory
58420|
58421|         PLIST_ENTRY
58421|         | p=DevExt->Cache.SnapShotHead.Flink;
58422|         while (
58422|         | p!=&DevExt->Cache.SnapShotHead ) {
58423|
58423|         | NumSnapShotsAdded++;
58424|
58424|         | pInternalSnapShot
58424|         | s=CONTAINING_RECORD(p,tInternalSnapShot,ListEntry);
58425|         // create a
58425|         | master if not already done
58426|
58426|         | pkSnapShotMaster myss = GetAMasterSnapShot(User,s);
58427|         // create a
58427|         | snapshot entry
58428|
58428|         | pkSnapShotEntry myse =
58428|         | GetSnapShotForMaster(myss,Volume,s);
58429|         p=p->Flink;
58430|     }
58431|
58432|         // Process the
58432|         | index file to rebuild the rbtree:
58433|         if (
58433|         | NT_SUCCESS(Status) ) {
58434|
58434|         | tIndexSector *indexSector; // not to be confused with
58434|         | 'GlobalIndexSector'.
58435|
58435|         | tIndexSector *IndexBlock;
58436|         ULONG
58436|         | sectorNumber = 0;
58437|
58437|         | FILE_STANDARD_INFORMATION FSInfo={0};
58438|         //
58438|         | IO_STATUS_BLOCK IoStatus;
58439|
58440|         | ASSERT(sizeof(tIndexSector)==SectorSize);
58441|
58442|         | LARGE_INTEGER L;
58443|         ULONG
58443|         | NumEntriesInCache = DevExt->Cache.PSManBitMapMaxSize;

```

```

58444|                                     signed long
    | NumSectorsLeftInBlock = 0;
58445|
58446|                                     // volume
    | already has a cache file. lets use its size
58447|                                     Status =
    | DevExt->Cache.CacheFile.Direct->getFileSizeInBytes(L);
58448|
58449|
    | if(NT_SUCCESS(Status)) {
58450|
    | ASSERT(L.QuadPart); // make sure not zero
58451|
    | L.QuadPart/=(GRANULE_SIZE); // get number of granules
58452|
    | ASSERT(L.HighPart==0); // cant handle things this large
58453|
    | Debug(DEBUG_DICT,("Rebuild: Reusing cache file size:
    | cache file is %d granules instead of %d
    | granules\n",L.LowPart,NumEntriesInCache));
58454|
    | NumEntriesInCache = L.LowPart;
58455|                                     } else {
58456|                                     // use
    | size retrieved from bitmap max size
58457|                                     Status
    | = STATUS_SUCCESS;
58458|                                     }
58459|
58460|                                     //
    | fixfixfix we need to do sanity checks to make sure we
    | dont
58461|                                     // use
    | mismatched files
58462|                                     // 1. Use
    | creation date/time to make sure files are matched
58463|                                     // 2. Check
    | NumEntries in index and make sure it matches NumEntries
    | in cache
58464|
58465|     #define LO_INDEXSECTORS_PER_BLOCK 128
58466|     #define HI_INDEXSECTORS_PER_BLOCK 1024
58467|
58468|
    | IO_SCSI_CAPABILITIES Caps={0};
58469|                                     ULONG
    | IndexSectorsPerBlock=LO_INDEXSECTORS_PER_BLOCK;
58470|
58471|                                     Status =
    | Sblo_GetCapabilities(DevExt->TargetDeviceObject,&Caps);

```

```

58472|
| if(NT_SUCCESS(Status)) {
58473|
| Debug(DEBUG_DICT,("Rebuild: MaxTransferSize=%d,
| MaxPages=%d\n",Caps.MaximumTransferLength,Caps.MaximumPh
| ysicalPages));
58474|
58475|
| if(Caps.MaximumTransferLength>=512*HI_INDEXSECTORS_PER_B
| LOCK) {
58476|
| IndexSectorsPerBlock = HI_INDEXSECTORS_PER_BLOCK;
58477|
| } else
| {
58478|
| IndexSectorsPerBlock = LO_INDEXSECTORS_PER_BLOCK;
58479|
| }
58480|
| } else {
58481|
| Debug(DEBUG_DICT,("Rebuild: Error %08x retrieving
| caps\n",Status));
58482|
| Status
| = 0;
58483|
| IndexSectorsPerBlock = LO_INDEXSECTORS_PER_BLOCK;
58484|
| }
58485|
58486|
| Debug(DEBUG_DICT,("Rebuild: Indexes per
| block=%d\n",IndexSectorsPerBlock*512));
58487|
58488|
| IndexBlock
| =
| (tIndexSector*)MemAllocatePoolWithTag(PagedPool,sizeof(t
| IndexSector)*(IndexSectorsPerBlock),TEMPTAG);
58489|
| if (
| IndexBlock ) {
58490|
| for (
| sectorNumber=0; sectorNumber<NumEntriesInCache;
| ++sectorNumber,++indexSector ) {
58491|
| IO_STATUS_BLOCK IoStatus;
58492|
| LARGE_INTEGER Location;
58493|
58494|
| if
| ( --NumSectorsLeftInBlock<=0 ) {
58495|
58496|
| NumSectorsLeftInBlock = min(IndexSectorsPerBlock,

```

```

    | NumEntriesInCache-sectorNumber);
58497|
    | indexSector = IndexBlock;
58498|
    | Location.QuadPart = (unsigned
    | __int64)SectorSize*sectorNumber;
58499|
58500|
    | START_STOPWATCH(Rebuild_ReadFromFile);
58501|
    | Status= PsmReadFromFile(
58502|
    | &DevExt->Cache.IndexFile,
58503|
    | &IoStatus,
58504|
    | IndexBlock,
58505|
    | (SectorSize*NumSectorsLeftInBlock),
58506|
    | &Location,
58507|
    | DevExt->DoDirectIO,
58508|
    | &DevExt->Cache.DirectAccessResource );
58509|
    | PAUSE_STOPWATCH(Rebuild_ReadFromFile);
58510|
58511|                                     }
58512|
58513|                                     if
    | ( NT_SUCCESS(Status) &&
    | IndexSectorIsValid(indexSector,sectorNumber,Status) ) {
58514|
    | START_STOPWATCH(Rebuild_AllocNode);
58515|
    | tTreeLeaf *Node = AllocNode();
58516|
    | PAUSE_STOPWATCH(Rebuild_AllocNode);
58517|
58518| #if DEBUG_VALIDATE_DIFF_GRANULES
58519|
    | if ( diffGranuleBuffer != NULL ) {
58520|
    | ValidateDiffGranule (indexSector, diffGranuleBuffer,
    | DevExt, sectorNumber);
58521|
    | }
58522| #endif /*DEBUG_VALIDATE_DIFF_GRANULES*/
58523|

```



```

    | BOOLEAN FreeTheNode = TRUE;
58524|
    | if ( Node ) {
58525|
    | int insertResult = 0;
58526|
    | ULARGE_INTEGER Key = {0};
58527|
    | pkSnapshotEntry p=NULL;
58528|
    | pDictionary Dict=NULL;
58529|
    | pkSnapshotMaster m=NULL;
58530|
    | pInternalSnapShot SnapShot;
58531|
58532|
    | Key.SnapShotPart =
    | indexSector->Info.DiskNode[0].SnapshotNumber;
58533|
    | Key.GranulePart =
    | indexSector->Info.DiskNode[0].DasdGranule;
58534|
58535|
    | // get a volume object based on the info stored in the
    | index
58536|
    | //VolObj =
    | GetVolumeObjectFromVolumeld(indexSector->Info.DiskNode[0
    | ].Volumeld);
58537|
58538|
    | // it is possible (and valid) for VolObj to be NULL.
    | this occurs when we have a snapshot for a
58539|
    | // volume that is not active. This can happen for
    | several reasons:
58540|
    | // 1. Volume is offline when in a cluster
58541|
    | // 2. Volume is not yet mounted.
58542|
    | // We will later fix up everything when/if the volume
    | comes online
58543|
58544|
    | // find an existing snapshot to insert node to
58545|
    | START_STOPWATCH(Rebuild_FindDict);
58546|

```

```

    | SnapShot =
    | FindSnapShotFromSequence(Volume,Key.SnapShotPart);
58547|
    | if ( SnapShot ) {
58548|
        | // active snapshot found
58549|
        | // get a master snapshot entry
58550|
        | m = GetAMasterSnapShot(User,SnapShot);
58551|
        | if ( m ) {
58552|
            | // get a snapshot entry (and dictionary) based on the
            | volume
58553|
            | p = GetSnapShotForMaster(m,Volume,SnapShot);
58554|
            | Dict= p->Dictionary;
58555|
            | }
58556|
        | } else {
58557|
            | // we have a node for a snapshot that has been deleted
58558|
            | GetDictionaryForVolume(Volume,Dict);
58559|
            | // FIXFIXFIX what to do when a node is deleted for a
            | volume that is
58560|
            | // not active????
58561|
            | }
58562|
        | PAUSE_STOPWATCH(Rebuild_FindDict);
58563|
58564|
58565|
        | START_STOPWATCH(Rebuild_Adding);
58566|
        | if ( Dict ) {
58567|
            | Node->Key = Key.QuadPart;
58568|
            | Node->Pos = indexSector->Info.DiskNode[0].CacheNode;
58569|
58570|
            | START_STOPWATCH(Rebuild_SettingBits);
58571|

```

```

    | // say the cache file entries are used
58572|
    | PsmBitPositionValidate
    | (DevExt->Cache.PSManBitMapBuffer, Node->Pos);
58573|
    | RtlSetBits ( DevExt->Cache.PSManBitMapBuffer,
    | Node->Pos, 1 );
58574|
    | InterlockedIncrement (
    | (PLONG)&DevExt->Cache.CurrentCacheFileSize );
58575|
    | PAUSE_STOPWATCH(Rebuild_SettingBits);
58576|
58577|
    | ASSERT(((pPersistentDictionary)Dict)->Shared);
58578|
58579|
    | START_STOPWATCH(Rebuild_AcquireExclusive);
58580|
    | MyAcquireResourceExclusiveLite (
    | &((pPersistentDictionary)Dict)->Shared->TreeResource,
    | TRUE );
58581|
    | PAUSE_STOPWATCH(Rebuild_AcquireExclusive);
58582|
    | __try {
58583|
    | START_STOPWATCH(Rebuild_InsertToTree);
58584|
    | if ( indexSector->Info.DiskNode[0].Flags &
    | PSM_INDEX_FLAG_VIRTUAL_WRITE ) {
58585|
    | pDictionary VirDict = 0;
58586|
58587|
    | GetDictionaryForVolume(Volume, VirDict, Key.SnapShotPart);
58588|
58589|
    | if ( VirDict ) {
58590|
    | //Debug(DEBUG_DICT,("Adding key=%016l64x pos=%08x to
    | virtual\n",Node->Key,Node->Pos));
58591|
    | // If combining virtual write trees, leave
    | Key.SnapShotPart as the sequence number
58592|
    | Debug(DEBUG_DICT,("Rebuild: Volume %08x: Adding key
    | %016l64x, pos %08x to combined virtual
    | tree\n",Volume,Node->Key,Node->Pos));
58593|

```

```

    | insertResult = rbtree_Insert(
58594|
    | &(((pPersistentDictionary)VirDict)->Shared->VirtualWrite
    | sTree),
58595|
    | Node);
58596|
58597|
    | if ( insertResult == 0 ) {
58598|
    | FreeTheNode = FALSE;
58599|
    | Profile("Reload: Virtual Write Nodes");
58600|
    | }
58601|
    | } else {
58602|
    | // no snapshot to apply this virtual write to. maybe it
    | was deleted.
58603|
    | Debug(DEBUG_DICT,("No dictionary to add key=%016l64x
    | pos=%08x to virtual\n",Node->Key,Node->Pos));
58604|
    | PsmBitPositionValidate
    | (DevExt->Cache.PSManBitMapBuffer, Node->Pos);
58605|
    | RtlClearBits ( DevExt->Cache.PSManBitMapBuffer,
    | Node->Pos, 1 );
58606|
    | InterlockedDecrement (
    | (PLONG)&DevExt->Cache.CurrentCacheFileSize );
58607|
    | insertResult = 0;
58608|
    | }
58609|
    | } else {
58610|
    | //Debug(DEBUG_DICT,("Rebuild: Volume %08x: Adding key
    | %016l64x, pos %08x to shared
    | tree\n",Volume,Node->Key,Node->Pos));
58611|
    | //Debug(DEBUG_DICT,("Adding key=%016l64x pos=%08x to
    | shared\n",Node->Key,Node->Pos));
58612|
    | insertResult = rbtree_Insert(
58613|
    | &((pPersistentDictionary)Dict)->Shared->Tree,
58614|

```

```

    | Node);
58615|
    | if ( insertResult == 0 ) {
58616|
    | FreeTheNode = FALSE;
58617|
    | Profile("Reload: Cached Data Nodes");
58618|
    | }
58619|
    | }
58620|
    | PAUSE_STOPWATCH(Rebuild_InsertToTree);
58621|
    | } __finally {
58622|
    | MyReleaseResourceForThreadLite (
    | &((pPersistentDictionary)Dict)->Shared->TreeResource );
58623|
    | }
58624|
58625|
    | if ( insertResult != 0 ) {
58626|
    | PsmBitPositionValidate
    | (DevExt->Cache.PSManBitMapBuffer, Node->Pos);
58627|
    | RtlClearBits ( DevExt->Cache.PSManBitMapBuffer,
    | Node->Pos, 1 );
58628|
    | InterlockedDecrement (
    | (PLONG)&DevExt->Cache.CurrentCacheFileSize );
58629|
58630|
    | LogTreeInsertionError ( Node, insertResult );
58631|
    | ++numberOfFailedInserts;
58632|     #ifdef DEBUG
58633|
    | DbgBreakPoint();
58634|     #endif
58635|
    | //status = STATUS_FILE_CORRUPT_ERROR;
58636|
    | } else {
58637|
    | ++numberOfNodesInserted;
58638|
    | if ( indexSector->Info.DiskNode[0].Flags &
    | PSM_INDEX_FLAG_VIRTUAL_WRITE ) {

```

```

58639|
    | ++numberOfVirtualWrites;
58640|
    | }
58641|
    | }
58642|
58643|
    | if (
    | Key.SnapShotPart>((pPersistentDictionary)Dict)->Shared->
    | HighestSequence ) {
58644|
    | ((pPersistentDictionary)Dict)->Shared->HighestSequence
    | = Key.SnapShotPart;
58645|
    | }
58646|
    | } else {
58647|
    | LogSnapshotEntryNotFound(m,Volume,sectorNumber);
58648|
    | Status = STATUS_FILE_CORRUPT_ERROR;
58649|
    | }
58650|
    | PAUSE_STOPWATCH(Rebuild_Adding);
58651|
    | if(FreeTheNode) {
58652|
    | START_STOPWATCH(Rebuild_FreeNode);
58653|
    | FreeNode(Node);
58654|
    | PAUSE_STOPWATCH(Rebuild_FreeNode);
58655|
    | }
58656|
    | } else {
58657|
    | Debug(DEBUG_DICT,( "!!! Cannot allocate rbtree node\n"
    | ));
58658|
    | Status = STATUS_INSUFFICIENT_RESOURCES;
58659|
    | }
58660|
    | }
    | else {
58661|
    | if ( !NT_SUCCESS(Status) ) {
58662|

```

```

    | if((Status==STATUS_DEVICE_OFF_LINE) ||
    | (Status==STATUS_DEVICE_BUSY)) {
58663|
    | // this can happen on cluster servers
58664|
    | Debug(DEBUG_DICT,("!!! Device went offline during
    | rebuild\n"));
58665|
    | } else {
58666|
    | // Corruption in the index file!
58667|
    | LogCorruptIndexSector ( sectorNumber, Status );
58668|
    | Status = STATUS_SUCCESS; // try reading rest of index
58669|
    | }
58670|
    | }
58671|                                     }
58672|
58673|                                     if
    | ( !NT_SUCCESS(Status) ) {
58674|
    | break; // for loop
58675|                                     }
58676|                                     } //
    | for
58677|
58678|
    | if(NT_SUCCESS(Status)) {
58679|                                     //
    | We need to get the dictionary of any snapshot on the
    | volume so we can address the rbtree
58680|                                     //
    | we know we must get one because we know we had
    | snapshots to insert for which we had to get one already
58681|
    | START_STOPWATCH(Rebuild_FreeDeadNodes);
58682|
    | pDictionary Dict;
58683|
    | GetDictionaryForVolume(Volume,Dict);
58684|
58685|                                     #if
    | SCAVENGESYNCHRONOUS
58686|
    | ((pPersistentDictionary)Dict)->Shared->LastDirtyKey =
    | ((pPersistentDictionary)Dict)->Shared->HighestKeyKnownCl
    | ean;

```

```

58687| | ((pPersistentDictionary)Dict)->Shared->RecycleNeeded =
| 1;
58688| | Debug(DEBUG_DICT,("pd::cleanup - Starting FreeDeadNodes
| - HighestClean: %8x|%08x LastDirty: %8x|%08x\n"
58689| | ,((pPersistentDictionary)Dict)->Shared->HighestKeyKnownC
| lean
58690| | ,((pPersistentDictionary)Dict)->Shared->LastDirtyKey
58691| | ));
58692| | FreeDeadNodes();
58693| | Debug(DEBUG_DICT,("pd::cleanup - Finished FreeDeadNodes
| - HighestClean: %8x|%08x LastDirty: %8x|%08x\n"
58694| | ,((pPersistentDictionary)Dict)->Shared->HighestKeyKnownC
| lean
58695| | ,((pPersistentDictionary)Dict)->Shared->LastDirtyKey
58696| | ));
58697| | #else
58698| | //TODO synchronize these variable changes with scavenge
| thread
58699| | //+
| | Test for kick the dog awake is needed if we let the
| | scavenger sleep
58700| | ((pPersistentDictionary)Dict)->Shared->LastDirtyKey =
| ((pPersistentDictionary)Dict)->Shared->HighestKeyKnownC
| lean;
58701| | ((pPersistentDictionary)Dict)->Shared->RecycleNeeded =
| 1;
58702| | #endif
58703| | PAUSE_STOPWATCH(Rebuild_FreeDeadNodes);
58704| |
58705| | } else
| {
58706| | //
| | since we are exiting, and the reference count will stay
| | at 0
58707| | //
| | lets cleanup

```



```

58708|
| DevExt->Cache.ReferenceCount+=NumSnapShotsAdded;
58709|
| RemoveVolumesFromSystem(Volume);
58710|
| ASSERT(DevExt->Cache.ReferenceCount==0);
58711|
| DevExt->Cache.ReferenceCount=0;
58712|                                     }
58713|
58714|
| FREE_POINTER(IndexBlock);
58715|                                     } else {
58716|
| Debug(DEBUG_DICT,("Out of memory for index block buffer
| for reading master snapshots\n"));
58717|                                     Status
| = STATUS_INSUFFICIENT_RESOURCES;
58718|                                     }
58719|                                     } else { // if
| NT_SUCCESS(Status)
58720|
| Debug(DEBUG_DICT,("Error %08x creating master
| snapshots\n",Status));
58721|                                     }
58722|
58723|
58724|                                     if (
| NT_SUCCESS(Status) ) {
58725|
| DevExt->Cache.ReferenceCount+=NumSnapShotsAdded;
58726|
| try_return(Status);
58727|                                     }
58728|                                     } else {
58729|                                     // no snapshots
| during rebuild, exit out
58730|                                     Status =
| STATUS_NOT_FOUND;
58731|                                     }
58732|                                     } else {
58733|                                     // not in rebuild,
| all resources have been acquired
58734|                                     // now exit
58735|                                     Status =
| STATUS_SUCCESS;
58736|
| DevExt->Cache.ReferenceCount++;
58737|
| try_return(NOTHING);

```

```

58738|         }
58739|     | FreeMemoryForVolume(Volume);
58740|         } else {
58741|     | Debug(DEBUG_DICT,("RebuildSnapShotsForVolume: Error
    | %08x allocating mem on volume %08x\n",Status,Volume));
58742|         }
58743|         if(!Rebuild) {
58744|     | DeleteFileObjectToSkip(DevExt->Cache.HeaderFile.FileObje
    | ct);
58745|     | DeleteFileObjectToSkip(DevExt->Cache.IndexFile.FileObjec
    | t);
58746|     | DeleteFileObjectToSkip(DevExt->Cache.CacheFile.FileObjec
    | t);
58747|         }
58748|         FreeHeader(Volume);
58749|     } else {
58750|     | Debug(DEBUG_DICT,("RebuildSnapShotsForVolume: Error
    | %08x loading header on volume %08x\n",Status,Volume));
58751|         }
58752|         if(!Rebuild) {
58753|     | Debug(DEBUG_DEVCON,("RebuildSnapShotsForVolume:
    | directio=%d, Setting to true\n",DevExt->DoDirectIO));
58754|         DevExt->DoDirectIO = TRUE;
58755|     | CloseFilesForVolume(Volume);
58756|         }
58757|     }
58758|     } else {
58759|         if ( (Status ==
    | STATUS_OBJECT_NAME_NOT_FOUND) || (Status ==
    | STATUS_OBJECT_PATH_NOT_FOUND) ) {
58760|     | Debug(DEBUG_DICT,("RebuildSnapShotsForVolume: Volume
    | %08x has no snapshots\n",Volume));
58761|         if ( !Rebuild ) {
58762|             // this is a new one
58763|             // Params are in MB,
    | convert to number of granules
58764|             ULONG ISC;
58765|             ULONG MSC;
58766|
58767|             GetCacheSizes( DevExt, ISC,
    | MSC );

```

```

58768|
58769|     | DevExt->Cache.PSManBitMapSize =
58770|     | ISC*((1024*1024)/GRANULE_SIZE);
58771|
58772|     | DevExt->Cache.PSManBitMapMaxSize =
58773|     | MSC*((1024*1024)/GRANULE_SIZE);
58774| #ifdef DEBUG
58775|         if(DevExt->DoDirectIO) {
58776|             // if we are doing
58777|             | directio, and no files exist, and we have snapshots
58778|             // something is
58779|             | screwed.
58780|             ASSERT(!DevExt->PSMed);
58781|             | ASSERT(IsListEmpty(&DevExt->Cache.SnapShotHead));
58782|             }
58783| #endif
58784|         // no need to do directio
58785|         | since no files exist.
58786|         DevExt->DoDirectIO = FALSE;
58787|         Status =
58788|         | CreateFilesForVolume(Volume,AbortEvent);
58789|         if ( NT_SUCCESS(Status) ) {
58790|             | Debug(DEBUG_DICT,("RebuildSnapShotsForVolume:
58791|             | CreateFilesForVolume succeeded, jumping back to
58792|             | DoOpen\n"));
58793|             goto DoOpen;
58794|         } else {
58795|             | Debug(DEBUG_DICT,("RebuildSnapShotsForVolume: Error
58796|             | %08x creating files on volume %08x\n",Status,Volume));
58797|             }
58798|         }
58799|         } else {
58800|             | Debug(DEBUG_DICT,("RebuildSnapShotsForVolume: Error
58801|             | %08x opening files on volume %08x\n",Status,Volume));
58802|             }
58803|         }
58804|         | ASSERT(IsListEmpty(&DevExt->Cache.SnapShotHead));
58805|         ASSERT(Status!=STATUS_SUCCESS);
58806|         ASSERT(DevExt->Cache.ReferenceCount==0);
58807|     } else {
58808|         // Reference count !=0
58809|         Status = STATUS_SUCCESS;
58810|         DevExt->Cache.ReferenceCount++;

```

```

58801|     }
58802| try_exit: NOTHING;
58803|
58804|     Debug(DEBUG_DICT,( "RebuildSnapShotsForVolume()
    | Number of nodes inserted in virtual trees =
    | %d\n", (unsigned long) numberOfVirtualWrites));
58805|     Debug(DEBUG_DICT,( "RebuildSnapShotsForVolume()
    | has inserted %lu nodes into the rbtree with status of
    | %08x\n", (unsigned long) numberOfNodesInserted, Status
    | ));
58806|
58807|     Debug(DEBUG_DICT,( "RebuildSnapShotsForVolume()
    | Number of nodes insertion failures = %d\n", (unsigned
    | long) numberOfFailedInserts));
58808|     Debug(DEBUG_DICT,( "RebuildSnapShotsForVolume()
    | returning 0x%08x\n", (unsigned long) Status ));
58809|
58810| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
58811|     Status = GetExceptionCode();
58812|     Debug(DEBUG_SFILTER, ("Exception %08x in
    | RebuildSnapShotsForVolume\n", Status));
58813| }
58814|
58815| if ( NT_SUCCESS(Status) ) {
58816|     if ( Rebuild ) {
58817|         // Avoid revert check if we booted 'nopsm'
    | or 'resetpsm'...
58818|         if ( QueryNoPsm() || QueryResetPsm() ) {
58819|
    | Debug(DEBUG_DICT, ("pd::RebuildSnapShotsForVolume:
    | skipping revert check due to nopsm/resetpsm\n"));
58820|         } else {
58821|             // Do not set Status to return value of
    | revert check.
58822|             // We do this on purpose, because we
    | don't want to indicate that
58823|             // the snapshot load failed. That is
    | bad!
58824|             START_STOPWATCH(Rebuild_RevertCheck);
58825|             RevertCheckAtVolumeMount(Volume);
58826|             PAUSE_STOPWATCH(Rebuild_RevertCheck);
58827|         }
58828|     }
58829|
58830|     // Do a SaveHeader here... this has the
    | side-effect of incrementing the dirty counter,
58831|     // so if a cluster failover happens, the other
    | machine can tell whether it needs to reload
58832|     // the index file.

```

```

58833|
58834|     NTSTATUS SaveHeaderStatus = SaveHeader(Volume);
58835|     if ( NT_SUCCESS(SaveHeaderStatus) ) {
58836|         | Debug(DEBUG_DICT,("pd::RebuildSnapShotsForVolume:
58837|         | Updated dirty counter in header\n"));
58838|     } else {
58839|         | Debug(DEBUG_DICT,("pd::RebuildSnapShotsForVolume:
58840|         | Error %08x updating dirty counter in
58841|         | header!\n",SaveHeaderStatus));
58842|         ASSERT(FALSE);
58843|     }
58844| }
58845| }
58846| #if DEBUG_VALIDATE_DIFF_GRANULES
58847| if ( diffGranuleBuffer != NULL ) {
58848|     FREE_POINTER(diffGranuleBuffer);
58849| }
58850| #endif /*DEBUG_VALIDATE_DIFF_GRANULES*/
58851| PAUSE_STOPWATCH(Rebuild_Total);
58852| UpdateGlobalStatus(PSM_IDLE);
58853| Debug(DEBUG_DICT,("pd::RebuildSnapShotsForVolume
58854| | returning %08x\n",Status));
58855| STOPWATCH_DUMPALL();
58856| STOPWATCH_RESETALL();
58857| return Status;
58858| }
58859| //-----
58860| | -----
58861| | -----
58862|
58863| typedef struct sZeroOut {
58864|     WCHAR *FileName;
58865|     LARGE_INTEGER Size;
58866| } tZeroOut;
58867|
58868| void ZeroOutFileThread ( void *ptr )
58869| {
58870|     tZeroOut *Zero=(tZeroOut*)ptr;
58871|     ASSERT(Zero);
58872|     if ( Zero ) {
58873|         | SbCreateAndFillFile(Zero->FileName,&Zero->Size,NULL,0x00
58874|         | ,FILLONWRITE_ALL);
58875|         MemFreePool(Zero);
58876|     }

```

```

58873|
58874|   PsTerminateSystemThread( 0 );
58875| }
58876|
58877| //-----
| -----
| -----
58878| // The purpose of this procedure is to release any
| resources
58879| // associated with a volume. This does NOT delete
| snapshots from
58880| // the system. This is called when a volume goes away
| (dismounted)
58881| // it should undo everything RebuildSnapShotsForVolume
| did.
58882|
58883| NTSTATUS PersistentDictionary::TearDownCacheForVolume(
| PDEVICE_OBJECT Volume )
58884| {
58885|   PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(Volume);
58886|   Debug(DEBUG_DICT,("pd::TearDownCacheForVolume:
| Volume=%08x\n",Volume));
58887|
58888|   if(!DevExt->InLoadUnload) {
58889|       // zero out index file to optimize load time
| after reboot
58890|       tZeroOut *Zero = (tZeroOut
| *)MemAllocatePoolWithTag(PagedPool,sizeof(tZeroOut),TEMP
| TAG);
58891|       if(Zero) {
58892|           Zero->Size = RtlEnlargedUnsignedMultiply (
| DevExt->Cache.PSManBitMapSize, SectorSize );
58893|           Zero->FileName =
| DevExt->Cache.IndexFile.FileName;
58894|
58895|           HANDLE TempHandle;
58896|
58897|           pmStartThread(
58898|
| (PKSTART_ROUTINE)ZeroOutFileThread, // IN
| PKSTART_ROUTINE StartRoutine,
58899|           (PVOID)Zero,
| // IN PVOID StartContext
58900|           &TempHandle //
| OUT PHANDLE ThreadHandle,
58901|           );
58902|           ZwClose(TempHandle);
58903|       }
58904|   }

```

```

58905|
58906|
58907|
58908|
58909|
58910|
58911|
58912|
58913|
58914| }
58915|
58916| //-----
58917|
58918| NTSTATUS
58919| {
58920|     PFILTERED_EXTENSION DevExt =
58921|     | GetFilteredExtension(Volume);
58922|     Debug(DEBUG_DICT, ("pd::ReleaseASnapShotForVolume:
58923|     | Volume=%08x, Snapshot=%08x, src=%d,
58924|     | crc=%d\n", Volume, Snapshot, Snapshot->ReferenceCount, DevEx
58925|     | t->Cache.ReferenceCount));
58926|     if ( --Snapshot->ReferenceCount==0 ) {
58927|         RemoveSnapShotFromList(SnapShot);
58928|         FreeSnapShot(&SnapShot);
58929|     }
58930|     if(!DevExt->InLoadUnload) {
58931|         SaveHeader(Volume);
58932|     }
58933|     if ( --DevExt->Cache.ReferenceCount==0 ) {
58934|         TearDownCacheForVolume(Volume);
58935|     }
58936|     return STATUS_SUCCESS;
58937| }
58938| #endif
58939|
58940|

```

```

58941| /*--- end of file perdict_rebuild.cpp ---*/
58942|
58943|
58944|
58945| File Listing: perdict_scavenge.cpp
58946|
58947| #include "precomp.h"
58948|
58949| //-----
| -----
58950| // definitions & variables global to delete functions
58951|
58952| #define SCAVENGESYNCHRONOUS 1
58953|
58954| #define MAXMOUTHFUL 200
58955|
58956| int ScavengeDebugFlag = 0;
58957|
58958| // Thread Control Macros
58959| #ifdef netware
58960| extern void *rbtree_Semaphore;
58961|
58962| #define DATA_REQUEST CPSemaphore(rbtree_Semaphore)
58963| #define DATA_RELEASE CVSemaphore(rbtree_Semaphore)
58964| #else
58965| #define DATA_REQUEST
58966| #define DATA_RELEASE
58967| #endif
58968|
58969|
58970|
58971| //-----
| -----
58972| // Checks if any snapshot still exists between 2 given
| node key values
58973| BOOLEAN
58974| PersistentDictionary::NodesDefunct( keyType
| PreviousKey, keyType CurrentKey)
58975| {
58976|     ULARGE_INTEGER myCurrentKey, myPreviousKey;
58977|     ULONG MyReferringLimit;
58978|     int i;
58979|
58980|     myCurrentKey.QuadPart = CurrentKey;
58981|     myPreviousKey.QuadPart = PreviousKey;
58982|
58983|     if ( myCurrentKey.GranulePart ==
| myPreviousKey.GranulePart ) {
58984|         MyReferringLimit=myPreviousKey.SnapShotPart;
58985|     } else {

```



```

58986|     MyReferringLimit=0;
58987| }
58988|
58989| // This logic can be streamlined if a table exists
    | of the viable snapshots in sequence
58990| // - which it may already do - need to check what
    | format Rob currently writes to disk
58991|
58992| PLIST_ENTRY p=DevExt->Cache.SnapShotHead.Flink;
58993| while ( p!=&DevExt->Cache.SnapShotHead ) {
58994|     pInternalSnapShot
    | s=CONTAINING_RECORD(p,tInternalSnapShot,ListEntry);
58995|
58996|     // Ignore the snapshot we are deleting as a
    | referrer because it's considered
58997|     // deleted even though it's still linked as
    | yet.
58998|
58999|     // FIXFIXFIX I THINK THIS WON'T WORK IF WE
    | REINSTATE SCAVENGE AT REBOOT !!!!!!!!!!!!!!!!!!!!!
    | check!!!!
59000|
59001|     if ( s != SnapShot ) {
59002|
59003|         if (
            | (s->Permanent.SequenceNumber<=myCurrentKey.SnapShotPart)
            | && (s->Permanent.SequenceNumber>MyReferringLimit) ) {
59004|             //then this referring snapshot is still
            | viable so node still required
59005|             if ( ScavengeDebugFlag ) {
59006|                 Debug(DEBUG_DICT,("Scavenge:
                    | %8x|%08x scope: |%x - %x Retained - referred to by %x
                    | \n"
59007|                                     ,
                    | myCurrentKey.GranulePart
59008|                                     ,
                    | myCurrentKey.SnapShotPart
59009|                                     ,
                    | MyReferringLimit+1
59010|                                     ,
                    | myCurrentKey.SnapShotPart
59011|                                     ,
                    | s->Permanent.SequenceNumber
59012|                                     ));
59013|             }
59014|             return 0;
59015|         }
59016|     }
59017|
59018|     p=p->Flink;

```

```

59019| }
59020| //no viables left using it so is defunct
59021| if ( ScavengeDebugFlag ) {
59022|     Debug(DEBUG_DICT,("Scavenge: %8x|%08x scope:
    | %x - %x Deleted - no referrers\n"
59023|         , myCurrentKey.GranulePart
59024|         , myCurrentKey.SnapShotPart
59025|         , MyReferringLimit+1
59026|         , myCurrentKey.SnapShotPart
59027|     ));
59028| }
59029| return 1;
59030| }
59031| //-----
    | -----
59032|
59033| tTreeLeaf *
59034| PersistentDictionary::ScanForDeadNode( keyType
    | PreviousKey, ULONG NumNodesToInspect)
59035| {
59036|     ULARGE_INTEGER myPreviousKey,CurrentKey;
59037|     tTreeLeaf *CurrentNode;
59038|
59039|     myPreviousKey.QuadPart = PreviousKey;
59040|     myPreviousKey.SnapShotPart++;
59041|     CurrentNode =
        | rbtree_SearchUpperBound(&Shared->Tree,
        | myPreviousKey.QuadPart);
59042|     myPreviousKey.SnapShotPart--;
59043|
59044| // DATA_REQUEST; //don't think I need this, maybe
    | if ever ported to Netware
59045|     while ( (NumNodesToInspect--) && (
        | (Shared->RecycleNeeded) ||
        | (Shared->HighestKeyKnownClean<Shared->LastDirtyKey) ) )
        | {
59046|         if ( CurrentNode == NULL ) {
59047|             //can only be no next if we've reached end
        | of tree so loop back to start
59048|             Debug(DEBUG_DICT,("Scavenge:
        | Recycling\n"));
59049|             Shared->RecycleNeeded = 0;
59050|             Shared->HighestKeyKnownClean =
        | myPreviousKey.QuadPart = 0;
59051|             CurrentNode =
        | rbtree_SearchUpperBound(&Shared->Tree,
        | myPreviousKey.QuadPart);
59052|             continue;
59053|         }
59054|         CurrentKey.QuadPart = CurrentNode->Key;

```

```

59055|     PreviousKey = myPreviousKey.QuadPart;
59056|     //added condition to not free nodes which have
    | the current highest key on the volume as they could
    | then be recached.
59057|     //This can lead to duplicate key errors
    | (c000022a) on a reboot which though benign (since the
    | original node
59058|     //wasn't ever gonna be needed again or we
    | wouldn't free it in the first place) lead to event log
    | messages and
59059|     //critical status on screens which would
    | mislead and worry the punters.
59060|     if ( (CurrentKey.SnapShotPart !=
    | Shared->HighestSequence) && (NodeIsDefunct(
    | PreviousKey, CurrentKey.QuadPart)) ) {
59061| //     DATA_RELEASE; //don't think I need this,
    | maybe if ever ported to Netware
59062|         return CurrentNode;
59063|     }
59064|     myPreviousKey = CurrentKey;
59065|     CurrentNode =
    | rbtree_GetNextInOrder(&Shared->Tree, CurrentNode);
59066|     // do i need some extra synchronising mechanism
    | here to ensure we do at LEAST a full cycle
59067|     Shared->HighestKeyKnownClean =
    | myPreviousKey.QuadPart;
59068| }
59069| // DATA_RELEASE; //don't think I need this, maybe
    | if ever ported to Netware
59070| return NULL;
59071| }
59072| //-----
    | -----
59073|
59074| void
59075| PersistentDictionary::FreeDeadNodes(void)
59076| {
59077|
59078| #if SCAVENGESYNCHRONOUS
59079| #else
59080|     //TODO determine these conditions
59081|     while ( appropriate allow exit ) {
59082| #endif
59083|
59084|         while ( Shared->RecycleNeeded ||
    | (Shared->HighestKeyKnownClean<Shared->LastDirtyKey) ) {
59085|             keyType KeyToDelete = 0;
59086|             tTreeLeaf *NodeToDelete =NULL;
59087|
59088|             MyAcquireResourceSharedLite(

```

```

    | &Shared->TreeResource, TRUE );
59089|     __try {
59090|         NodeToDelete =
    | ScanForDeadNode(Shared->HighestKeyKnownClean,
    | MAXMOUTHFUL);
59091|
59092|         if ( NodeToDelete != NULL ) {
59093|             //Only now that I know there is a
    | node can I get the key - and I must get it before I
    | release
59094|             //the shared lock as the node may
    | have moved by the time I get the exclusive lock!!!
59095|             KeyToDelete = NodeToDelete->Key;
59096|         }
59097|     } __finally {
59098|         //allow other processes to alter tree
    | structure between mouthfuls
59099|         MyReleaseResourceForThreadLite (
    | &Shared->TreeResource );
59100|     }
59101|
59102|     if ( NodeToDelete != NULL ) {
59103|         MyAcquireResourceExclusiveLite(
    | &Shared->TreeResource, TRUE );
59104|         __try {
59105|             | FreeNodeAndBits(rbtrees_Delete(&Shared->Tree,
    | KeyToDelete));
59106|             //Now switch back to shared mode
    | for the next search
59107|         } __finally {
59108|             MyReleaseResourceForThreadLite (
    | &Shared->TreeResource );
59109|         }
59110|     }
59111| }
59112|
59113| #if SCAVENGESYNCHRONOUS
59114| #else
59115| }
59116| #endif
59117|
59118| }
59119|
59120|
59121| //-----
    | -----
59122|
59123| ErrorCode
59124| PersistentDictionary::cleanup(void)

```

```

59125| {
59126|     NTSTATUS Status;
59127|
59128|     Debug(DEBUG_DICT,("pd::cleanup:
    | this=%08x\n",this));
59129|     __try {
59130|         tTreeLeaf *Node=NULL;
59131|
59132|         pPersistentDictionary p=ListHead;
59133|         pPersistentDictionary Prev=NULL;
59134|
59135|         while ( p ) {
59136|             if ( p == this ) {
59137|                 break;
59138|             }
59139|             Prev=p;
59140|             p=p->Next;
59141|         }
59142|
59143|         ASSERT(p);
59144|         if ( p ) {
59145|             if ( Prev ) {
59146|                 Prev->Next = Next;
59147|             } else {
59148|                 ListHead = Next;
59149|             }
59150|         }
59151|
59152|         RemoveVirtualWritesFromTree (FALSE);
59153|
59154|         ASSERT(Shared->Count>0);
59155|
59156|         Shared->Count--;
59157|         Debug(DEBUG_DICT,("pd::cleanup: (dict=%08x)
    | Decrement Shared->Count to %08x\n",
59158|             this,
59159|             Shared->Count));
59160|
59161|         if ( Shared->Count==0 ) {
59162|             Debug(DEBUG_DICT,("pd::cleanup -
    | Acquire\n"));
59163|             MyAcquireResourceExclusiveLite(
    | &Shared->TreeResource, TRUE );
59164|             Debug(DEBUG_DICT,("pd::cleanup - Freeing
    | nodes\n"));
59165|             __try {
59166|                 Node=Shared->Tree.HeadLeaf;
59167|                 while ( Node!=Shared->Tree.TailLeaf ) {
59168|                     FreeNodeAndBits(
    | rbtree_Delete(&Shared->Tree, Node->Key) );

```

```

59169|         Node = Shared->Tree.HeadLeaf;
59170|     }
59171| } __finally {
59172|     Debug(DEBUG_DICT,("pd::cleanup - init
    | tree\n"));
59173|     // reinit tree...
59174|     rbtree_Init( &Shared->Tree );
59175|     Debug(DEBUG_DICT,("pd::cleanup -
    | Release\n"));
59176|     MyReleaseResourceForThreadLite (
    | &Shared->TreeResource );
59177| }
59178|
59179|
59180|     Debug(DEBUG_DICT,("pd::cleanup - Delete
    | resource\n"));
59181|     // remove resource
59182|     ExDeleteResourceLite( &Shared->TreeResource
    | );
59183|     Debug(DEBUG_DICT,("pd::cleanup -
    | Deregister\n"));
59184|     pmDeRegisterObject(&Shared->TreeResource);
59185|     Debug(DEBUG_DICT,("pd::cleanup - Free
    | mem\n"));
59186|     if ( Shared->Map ) {
59187|         FREE_POINTER(Shared->Map);
59188|     }
59189|     FREE_POINTER(Shared);
59190| } else {
59191|
59192|     if(!DevExt->InLoadUnload) {
59193| #if SCAVENGESYNCHRONOUS
59194|         Shared->LastDirtyKey =
    | Shared->HighestKeyKnownClean;
59195|         Shared->RecycleNeeded = 1;
59196|         Debug(DEBUG_DICT,("pd::cleanup -
    | Starting FreeDeadNodes - HighestClean: %8x|%08x
    | LastDirty: %8x|%08x\n"
59197|         | ,Shared->HighestKeyKnownClean
59198|         | ,Shared->LastDirtyKey
59199|         ));
59200|         FreeDeadNodes();
59201|         Debug(DEBUG_DICT,("pd::cleanup -
    | Finished FreeDeadNodes - HighestClean: %8x|%08x
    | LastDirty: %8x|%08x\n"
59202|         | ,Shared->HighestKeyKnownClean
59203|         | ,Shared->LastDirtyKey
59204|         ));

```

```

59205|  #else
59206|      //TODO synchronize these variable
| changes with scavenge thread
59207|      //+ Test for kick the dog awake is
| needed if we let the scavenger sleep
59208|      Shared->LastDirtyKey =
| Shared->HighestKeyKnownClean;
59209|      Shared->RecycleNeeded = 1;
59210|  #endif
59211|  } // !in mount
59212|  }
59213|
59214|  if ( (SnapShot) &&
| (SnapShot->Permanent.SequenceNumber!=0) ) {
59215|      ReleaseASnapShotForVolume(Volume,SnapShot);
59216|  }
59217|
59218|  /*
59219|      if ( (GhostBusterTime == FALSE) &&
| (!DevExt->InLoadUnload)) {
59220|          Status = SaveHeader(Volume);
59221|          if ( !NT_SUCCESS(Status) ) {
59222|              Debug(DEBUG_DICT,("Error %08x saving
| header\n",Status));
59223|              // dont exit, as we cant do anything
| about it anyway so lets cleanup
59224|              // as much as we can
59225|              }
59226|          }
59227|      */
59228|
59229|
59230|      Status = STATUS_SUCCESS;
59231|  } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
59232|      Status = GetExceptionCode();
59233|      Debug(DEBUG_DICT,("Exception %08x in
| pd::cleanup\n",Status));
59234|  }
59235|
59236|  return Status;
59237| }
59238|
59239| /*--- end of file perdict_scavenge.cpp ---*/
59240|
59241|
59242|
59243| File Listing: perdict_search.cpp
59244|
59245| #include "precomp.h"

```

```

59246|
59247| // note: define DO_ALL_IO in precomp.h do get all io
    | related functions
59248| // otherwise only io in this file will get printed
59249|
59250| //-----
    | -----
59251|
59252| ErrorCode
59253| PersistentDictionary::searchAndDeleteSingle(
59254|
    | PFILTERED_EXTENSION DevExt,
59255|
    | ULARGE_INTEGER Sector )
59256| {
59257|     sTreeLeaf *Node;
59258|     NTSTATUS Status=STATUS_NOT_FOUND;
59259|
59260|     Profile("pd::searchAndDeleteSingle");
59261|     Debug(DEBUG_DICT,("pd::searchAndDeleteSingle\n"));
59262|     ASSERT(GlobalSystemProcessId ==
    | PsGetCurrentProcess());
59263|     __try {
59264|
59265|         MyAcquireResourceExclusiveLite (
    | &Shared->TreeResource, TRUE );
59266|         __try {
59267|             ULARGE_INTEGER TreeKey;
59268|             unsigned __int64 Granule = Sector.QuadPart
    | / SECTORS_PER_GRANULE;
59269|             ASSERT ( (Granule >> 32) == 0 );
59270|             TreeKey.GranulePart = (ULONG) Granule;
59271|             TreeKey.SnapShotPart = GetSequenceNumber();
59272|             if ( (Node =
    | rbtree_Delete(&Shared->Tree,TreeKey.QuadPart))!=NULL )
    | {
59273|                 FreeNodeAndBits(Node);
59274|                 Status = STATUS_SUCCESS;
59275|             }
59276|         } __finally {
59277|             MyReleaseResourceForThreadLite (
    | &Shared->TreeResource );
59278|         }
59279|     } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
59280|         Status = GetExceptionCode();
59281|         Debug(DEBUG_DICT,("Exception %08x in
    | pd::searchAndDeleteSingle\n",Status));
59282|     }
59283|

```



```

59284|   return Status;
59285| }
59286|
59287| //-----
    | -----
59288|
59289|
59290| bool EnableSearchMultipleDebug = false;
59291|
59292| ErrorCode
59293| PersistentDictionary::searchMultiple (
59294|     | PFILTERED_EXTENSION    DevExt,
59295|                                     ULARGE_INTEGER
    | Starting,
59296|                                     ULONG
    | Count,
59297|                                     ULONG
    | &CountDid,
59298|                                     PRTL_BITMAP
    | BitMap,
59299|                                     ULARGE_INTEGER
    | DataSize,
59300|                                     PVOID
    | VirtualDataPointer,
59301|                                     ULONG
    | Flags )
59302| {
59303|   NTSTATUS Status=STATUS_SUCCESS;
59304|   BOOLEAN Read=FALSE;
59305|   ULONG FirstInMem=0;
59306|   Profile("pd::searchMultiple");
59307|
59308|   ASSERT(GlobalSystemProcessId ==
    | PsGetCurrentProcess());
59309|   if ( Count==0 ) {
59310|       Debug(DEBUG_DICT,("pd::searchMultiple:
    | Count==0\n"));
59311|       return STATUS_SUCCESS;
59312|   }
59313|
59314|   __try {
59315|
59316|       // VirtualDataPointer contains the data as read
    | from the
59317|       // head drive.. it is not in GRANULE_SIZE
    | offsets or counts
59318|
59319|       CountDid = 0;
59320|       if ( BitMap ) {

```

```

59321|         PsmBitMapValidate (BitMap);
59322|         RtlClearAllBits(BitMap);
59323|     }
59324|
59325|     char *NextReadPoint = (char
    | *)VirtualDataPointer;
59326|     ULONG SectorsRemaining = Count;
59327|     ULARGE_INTEGER CurrentSector = Starting;
59328|     ULONG CurrentGranule =
    | (ULONG)(Starting.QuadPart / SECTORS_PER_GRANULE);
59329|     ULONG StartOffsetIntoGranule =
    | (ULONG)(Starting.QuadPart % SECTORS_PER_GRANULE);
59330|     tTreeLeaf *Node = 0;
59331|     ULARGE_INTEGER Key = {0};
59332|
59333|     while ( SectorsRemaining>0 &&
    | NT_SUCCESS(Status) ) {
59334|         // First figure out which sectors we need
    | to process within this granule.
59335|         // We need to know which sector to start at
    | (StartOffsetIntoGranule)
59336|         // and how many sectors to read
    | (SectorsToRead).
59337|
59338|         ULONG SectorsToRead= 0; // Sectors to
    | read in this granule, starting at given offset
59339|         if ( SectorsRemaining +
    | StartOffsetIntoGranule > SECTORS_PER_GRANULE ) {
59340|             SectorsToRead = SECTORS_PER_GRANULE -
    | StartOffsetIntoGranule;
59341|         } else {
59342|             SectorsToRead = SectorsRemaining;
59343|         }
59344|
59345| #if 0
59346|         if (
    | !NeedsCaching(CurrentSector,SectorsToRead) ) {
59347| #endif
59348|
59349|         // Now that we know what we are
    | supposed to read for this granule,
59350|         // we need to determine WHERE to get
    | the granule from: cache or dasd.
59351|
59352|         Key.SnapShotPart = GetSequenceNumber();
59353|         Key.GranulePart = CurrentGranule;
59354|
59355|         MyAcquireResourceSharedLite (
    | &Shared->TreeResource, TRUE );
59356|         if ( Flags & DICT_FLAG_VIRTUAL_IO ) {

```

```

59357|          // check to see if it has a virtual
    | write
59358|          Node = rbtree_Search (
    | &(Shared->VirtualWritesTree), Key.QuadPart );
59359|          if ( Node ) {
59360| #if DO_ALL_IO
59361|          if ( EnableSearchMultipleDebug
    | ) {
59362|          | Debug(DEBUG_DICT,("pd::searchMultiple: rbtree_Search[1]
    | found node in virtual tree: Key=%016l64x,
    | Pos=%08x\n",Node->Key,Node->Pos));
59363|          }
59364| #endif
59365|          } else {
59366|          // no virtual write, see if its
    | in the cache
59367|          Node = rbtree_SearchUpperBound(
    | &Shared->Tree, Key.QuadPart);
59368| #if DO_ALL_IO
59369|          if ( Node &&
    | EnableSearchMultipleDebug ) {
59370|          | Debug(DEBUG_DICT,("pd::searchMultiple:
    | rbtree_SearchUpperBound[2] found node in shared tree:
    | Key=%016l64x, Pos=%08x\n",Node->Key,Node->Pos));
59371|          }
59372| #endif
59373|          }
59374|          } else {
59375|          Node = rbtree_SearchUpperBound(
    | &Shared->Tree, Key.QuadPart);
59376| #if DO_ALL_IO
59377|          if ( Node &&
    | EnableSearchMultipleDebug ) {
59378|          | Debug(DEBUG_DICT,("pd::searchMultiple:
    | rbtree_SearchUpperBound[3] found node in shared tree:
    | Key=%016l64x, Pos=%08x\n",Node->Key,Node->Parent));
59379|          }
59380| #endif
59381|          }
59382|
59383|          if ( Node ) {
59384|          IO_STATUS_BLOCK IoStatus;
59385|          ULARGE_INTEGER FoundKey;
59386|
59387|          MyReleaseResourceForThreadLite (
    | &Shared->TreeResource );
59388|

```

```

59389|             FoundKey.QuadPart = Node->Key;
59390|
59391| #ifdef DEBUG
59392|             if ( !(Flags &
59393| | DICT_FLAG_VIRTUAL_IO) ) {
59394|                 ASSERT ( FoundKey.QuadPart >=
59395| | Key.QuadPart );
59396|             }
59397| #endif
59398|             // Need to determine whether this
59399| | is really the same granule.
59400|             if ( Key.GranulePart ==
59401| | FoundKey.GranulePart ) {
59402|                 // Found this node in the
59403| | dictionary, so read from cache
59404| #if DO_ALL_IO
59405|                 if ( EnableSearchMultipleDebug
59406| | ) {
59407|                     Debug(DEBUG_DICT,("pd::searchMultiple: %08l64x in cache
59408| | at pos %08x\n",Key,Node->Pos));
59409|                 }
59410| #endif
59411|                 if ( BitMap ) {
59412|                     ULONG NumSectors = (ULONG)
59413| | (CurrentSector.QuadPart - Starting.QuadPart);
59414|                     PsmBitRangeValidate
59415| | (BitMap, NumSectors, SectorsToRead);
59416|                     RtlSetBits( BitMap,
59417| | NumSectors, SectorsToRead );
59418|                 }
59419|                 CountDid+=SectorsToRead;
59420|
59421|                 if ( VirtualDataPointer ) {
59422|                     LARGE_INTEGER Location;
59423|                     Location.QuadPart =
59424| | ((unsigned __int64)Node->Pos*SECTORS_PER_GRANULE +
59425| | StartOffsetIntoGranule) * (unsigned __int64)SectorSize;
59426|
59427|                     | ASSERT(Node->Pos!=INVALID_POSITION_VALUE);
59428|
59429|                     | ASSERT(Node->Pos<=DevExt->Cache.PSManBitMapSize);
59430|
59431|                     Status = PsmReadFromFile(
59432| | &DevExt->Cache.CacheFile,
59433|                     &IoStatus,
59434|                     NextReadPoint,

```

```

59423|                SectorsToRead *
    | SectorSize,
59424|                &Location,
59425|                DevExt->DoDirectIO,
59426|
    | &DevExt->Cache.DirectAccessResource );
59427|
59428| #ifdef DEBUG_REVERT
59429|                Debug(DEBUG_DICT,(
59430|
    | "pd::searchMultiple: PsmReadFromFile returned 0x%08x;
    | CacheHandle=%p, ReadPoint=%p, Location=0x%l64x\n",
59431|                Status,
59432|
    | DevExt->Cache.CacheFile.FileHandle,
59433|
    | NextReadPoint,
59434|
    | Location.QuadPart));
59435| #endif /* DEBUG_REVERT */
59436|                }
59437|                } else {
59438|                // not in tree..
59439|                }
59440|                } else {
59441|                // not in tree..
59442|                MyReleaseResourceForThreadLite (
    | &Shared->TreeResource );
59443| #if DO_ALL_IO
59444|                if ( EnableSearchMultipleDebug ) {
59445|
    | Debug(DEBUG_DICT,("pd::searchMultiple: rbtree node not
    | found for search key = %016l64x\n",Key.QuadPart));
59446|                }
59447| #endif
59448|                }
59449| #if 0
59450|                } else { // free space
59451|                Debug(DEBUG_DICT,("pd::searchMultiple:
    | %08l64x is free space\n",Key));
59452|                if ( BitMap ) {
59453|                ULONG NumSectors = (ULONG)
    | (CurrentSector.QuadPart - Starting.QuadPart);
59454|                PsmBitRangeValidate (BitMap,
    | NumSectors, SectorsToRead);
59455|                RtlSetBits( BitMap, NumSectors,
    | SectorsToRead );
59456|                }
59457|                CountDid+=SectorsToRead;
59458|                }

```

```

59459| #endif
59460|     CurrentSector.QuadPart = ++CurrentGranule *
    | (unsigned __int64)SECTORS_PER_GRANULE;
59461|     StartOffsetIntoGranule = 0;    // for all
    | subsequent granules (if any) start at 0
59462|     SectorsRemaining -= SectorsToRead;
59463|     NextReadPoint += SectorSize *
    | SectorsToRead;
59464|     }
59465| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
59466|     Status = GetExceptionCode();
59467|     Debug(DEBUG_DICT,("Exception %08x in
    | pd::searchMultiple\n",Status));
59468| }
59469|
59470| if ( (!Status) && (!CountDid) ) {
59471|     // if we didn't do anything, say so..
59472|     Status = STATUS_NOT_FOUND;
59473| } else {
59474|     #if DO_ALL_IO
59475|         if ( EnableSearchMultipleDebug ) {
59476|             Debug(DEBUG_DICT,("pd::searchMultiple
    | %08x %08l64x %08x %08x %08x %08l64x %08x, %08x\n",
59477|                 DevExt->DeviceObject,
59478|                 Starting,
59479|                 Count,
59480|                 CountDid,
59481|                 BitMap,
59482|                 DataSize,
59483|                 VirtualDataPointer,
59484|                 Flags));
59485|         }
59486|     #endif /*DO_ALL_IO*/
59487| }
59488|
59489| return Status;
59490| }
59491|
59492| //-----
    | -----
59493| ErrorCode
59494| PersistentDictionary::searchAndInsertMultiple(
59495|     | PFILTERED_EXTENSION DevExt,
59496|     | ULARGE_INTEGER    Starting,
59497|                                     ULONG
    | Count,
59498|                                     ULONG

```

```

    | &CountDid,
59499|
    | PRTL_BITMAP      BitMap,
59500|
    | ULARGE_INTEGER   DataSize,
59501|                                PCVOID
    | DataPointer,
59502|
    | pDictSiblingInfo  SiblingInfo,
59503|                                ULONG
    | Flags )
59504| {
59505|     pPerSiblingInfo Info =
    | (pPerSiblingInfo)SiblingInfo;
59506|     ULONG Bit=0;
59507|     NTSTATUS Status=0;
59508|     Profile("pd::searchAndInsertMultiple");
59509|
59510|     ASSERT(GlobalSystemProcessId ==
    | PsGetCurrentProcess());
59511|
59512| #ifdef DEBUG
59513|     // maybe we don't need DevExt parameter? Here's an
    | experiment to find out...
59514|     if ( DevExt != this->DevExt ) {
59515|         static bool alreadyAsserted = false; //
    | don't assert over and over again, in case we do find
    | mismatching DevExt
59516|         if ( !alreadyAsserted ) {
59517|             alreadyAsserted = true;
59518|             ASSERT(DevExt == this->DevExt);
59519|         }
59520|     }
59521| #endif /*DEBUG*/
59522|
59523|     if ( Count==0 ) {
59524|         #if DO_ALL_SEARCH
59525|
    | Debug(DEBUG_DICT,("pd::searchAndInsertMultiple Count==0
    | \n"));
59526|         #endif /*DO_ALL_SEARCH*/
59527|
    | return STATUS_SUCCESS;
59528|     }
59529|
59530|
59531|     // on entry DataPointer is original data from disk
59532|     // we need to save it to our cache file
59533|
59534|     // for simplicity's sake we are only going to allow
    | Starting to be a multiple

```

```

59535| // of GRANULE_SIZE as the higher level code has
| already read the data
59536| // from disk using the granule offset
59537|
59538| ASSERT((Starting.QuadPart %
| SECTORS_PER_GRANULE)==0);
59539|
59540| __try {
59541|     const CHAR *NextWritePosition = (const
| char*)DataPointer;
59542|     ULONG DataChecksum = 0;
59543|     ULONG SnapshotNumber = 0;
59544|     ULONG CurrentGranule =
| (ULONG)(Starting.QuadPart / SECTORS_PER_GRANULE);
59545|     ULONG SectorOffsetIntoGranule =
| (ULONG)(Starting.QuadPart % SECTORS_PER_GRANULE);
59546|     ULONG SectorsRemaining = Count;
59547|     tTreeLeaf *Node = 0;
59548|     ULARGE_INTEGER Key={0};
59549|     ULARGE_INTEGER WorkingKey;
59550|
59551|     // If we are inserting into the shared virtual
| write tree,
59552|     // we need to make the snapshot part of the
| node's key be
59553|     // specifically for this snapshot, not the
| highest sequence number
59554|     // as we do for the normal shared tree.
59555|     ULONG VirtualWriteSequence =
| GetSequenceNumber();
59556|
59557|     ASSERT(DataSize.QuadPart>=Count*SectorSize);
59558|     ASSERT(DataPointer!=NULL);
59559|
59560|     if ( Info->AlreadyHandled ) {
59561|         // we have 1 dictionary per volume,
59562|         // the higher level code is going through
59563|         // all the snaps in the system, tell it to
59564|         // quit scanning..
59565|
| //Debug(DEBUG_DICT,("pd::searchAndInsertMultiple
| AlreadyHandled \n"));
59566|         try_return (Status = STATUS_END_OF_FILE);
59567|     }
59568|
59569| #if DO_ALL_IO
59570|     Debug(DEBUG_DICT,("pd::searchAndInsertMultiple
| %08x %08I64x %08x %08x %08x %08I64x %08x %08x
| %08x\n",DevExt->DeviceObject,Starting,Count,CountDid,Bit
| Map,DataSize,DataPointer,SiblingInfo,Flags));

```



```

59571| #endif
59572|
59573|     CountDid=0;
59574|
59575|     if ( BitMap ) {
59576|         PsmBitMapValidate (BitMap);
59577|         RtlClearAllBits(BitMap);
59578|     }
59579|
59580|     // Need to check for copy-on-write only if
    | snapshots exist.
59581|     // Getting here means at least one snapshot
    | does exist.
59582|
59583|
59584|     Key.QuadPart = 0;
59585|     // attach to latest snapshot for this volume
    | only
59586|     SnapshotNumber = Key.SnapShotPart =
    | Shared->HighestSequence; //Header->HighestSnapNumber;
    | //FindHighestSnapNumber();
59587|
59588|     ASSERT(Key.SnapShotPart>0);
59589|
59590|     // For each granule that this write will touch,
    | perform copy-on-write
59591|     // if it hasn't already been done.
59592|
59593|     while ( SectorsRemaining>0 &&
    | NT_SUCCESS(Status) ) {
59594|         ULONG SectorsToSkip = 0;
59595|         if ( SectorsRemaining +
    | SectorOffsetIntoGranule > SECTORS_PER_GRANULE ) {
59596|             SectorsToSkip = SECTORS_PER_GRANULE -
    | SectorOffsetIntoGranule;
59597|         } else {
59598|             SectorsToSkip = SectorsRemaining;
59599|         }
59600|
59601|         Key.GranulePart = CurrentGranule;
59602|
59603|         // not in any snapshots, so lets add it
59604|         Node = AllocNode();
59605|         if ( Node ) {
59606|             ULONG Added=0;
59607|             IO_STATUS_BLOCK IoStatus;
59608|             LARGE_INTEGER Location;
59609|             ULONG DontFree=0;
59610|
59611|             Node->Pos = INVALID_POSITION_VALUE;

```

```

59612|         MyAcquireResourceExclusiveLite (
| &Shared->TreeResource, TRUE );
59613|         if ( Flags & DICT_FLAG_VIRTUAL_IO ) {
59614|             ULARGE_INTEGER VKey=Key;
59615|             VKey.SnapShotPart =
| VirtualWriteSequence;
59616|             Node->Key = VKey.QuadPart;
59617|             Added =
| rbtree_Insert(&(Shared->VirtualWritesTree),Node)==0;
59618|             #if DO_ALL_SEARCH
59619|             | Debug(DEBUG_DICT,("pd::searchAndInsertMultiple -
| VirtualWrite: key=%016l64x insert:
| added=%x\n",VKey.QuadPart,Added));
59620|             #endif /*DO_ALL_SEARCH*/
59621|             if ( !Added ) {
59622|                 // ok, must already be in the
| tree, lets update
59623|                 // the data in the cache file
| with the now new
59624|                 // data
59625|                 FreeNode(Node);
59626|                 | Node=rbtree_Search(&(Shared->VirtualWritesTree),VKey.Qua
| dPart);
59627|                 if ( Node ) {
59628|                 | ASSERT(Node->Pos!=INVALID_POSITION_VALUE);
59629|                 Added = TRUE;
59630|                 DontFree = TRUE;
59631|                 } else {
59632|                 // shouldnt happen
59633|                 ASSERT(FALSE);
59634|                 }
59635|                 }
59636|                 } else {
59637|                 Node->Key = Key.QuadPart;
59638|                 Added =
| rbtree_Insert(&Shared->Tree,Node)==0;
59639|                 }
59640|
59641|                 WorkingKey.QuadPart = Node->Key;
59642|
59643|                 if ( Added ) {
59644|                 if ( Flags & DICT_FLAG_VIRTUAL_IO )
| {
59645|                 if (
| Node->Pos==INVALID_POSITION_VALUE ) {
59646|                 Node->Pos =
| GetNextCacheLocation(1);

```

```

59647|         }
59648|     } else {
59649|         Node->Pos =
59650|         | GetNextCacheLocation(1);
59651|     }
59652|     if (
59653|         | Node->Pos!=INVALID_POSITION_VALUE ) {
59654|         ULONG SavedPos = Node->Pos;
59655|         // release this while we do the
59656|         | write or a deadlock will occur.
59657|         // we have a valid value at
59658|         | this time, and unless an error occurs
59659|         // we dont have to worry about
59660|         | reacquiring it.
59661|         MyReleaseResourceForThreadLite
59662|         | ( &Shared->TreeResource );
59663|         // Do !!!!!NOT!!!!!! access
59664|         | Node from this point ON
59665|         // unless you like random
59666|         | corruption
59667|         Location.QuadPart = (unsigned
59668|         | __int64)SavedPos * GRANULE_SIZE;
59669|         ASSERT(Key.SnapShotPart>0);
59670|         #if DO_ALL_IO
59671|         | Debug(DEBUG_DICT,("pd::searchAndInsertMultiple: Writing
59672|         | %08l64x to cache file position %08x\n",Key,SavedPos));
59673|         #endif
59674|         | ASSERT(SavedPos<=DevExt->Cache.PSManBitMapSize);
59675|         #ifdef DEBUG
59676|         #if DO_ALL_SEARCH
59677|         if ( Flags &
59678|         | DICT_FLAG_VIRTUAL_IO ) {
59679|         | Debug(DEBUG_DICT,("pd::searchAndInsertMultiple -
59680|         | VirtualWrite: saving pos=%08x, key=%016l64x,
59681|         | this->Flags=%08x,
59682|         | SnapShotFlags=%08x\n",SavedPos,WorkingKey.QuadPart,this-
59683|         | >Flags,GetSnapShotFlags()));
59684|         }
59685|         #endif /*DO_ALL_SEARCH*/
59686|         #endif /*DEBUG*/
59687|         // write to cache file

```

```

59679|                Status = PsmWriteToFile(
59680|                    &DevExt->Cache.CacheFile,
59681|                    &IoStatus,
59682|                    (PVOID)NextWritePosition,
59683|                    GRANULE_SIZE,
59684|                    &Location,
59685|                    DevExt->DoDirectIO,
59686|                    | &DevExt->Cache.DirectAccessResource );
59687|
59688|                if ( NT_SUCCESS(Status) ) {
59689|                    // Need to save to index
                    | file whether we have a temporary or persistent
                    | snapshot.
59690|                    // This is due to the
                    | lookahead algorithm. In other words, persistent
                    | snapshots
59691|                    // inherit changes made to
                    | subsequent temporary snapshots. Even when the
                    | temporary
59692|                    // snapshot goes away, we
                    | need the changes it made to the index file for
                    | persistent
59693|                    // snapshots. The
                    | scavenger will clean up index entries created by
                    | temporary snapshots
59694|                    // with no preceeding
                    | persistent snapshots.
59695|
59696|                    ULONG Virtuallo = (Flags &
                    | DICT_FLAG_VIRTUAL_IO) ? TRUE : FALSE;
59697|                    unsigned char SnapShotFlags
                    | = (unsigned char) GetSnapShotFlags();
59698|                    ULONG ReadWritePersistent =
                    | (SnapShotFlags & PSM_SS_BIT_VIRTUAL_WRITES_PERSISTENT)
                    | ? TRUE : FALSE;
59699|                    if ( ReadWritePersistent )
                    | {
59700|                        ASSERT
                    | (PSM_SS_IsPersistent(SnapShotFlags));
59701|                    }
59702|
59703|                    if ( (Virtuallo &&
                    | ReadWritePersistent) || !Virtuallo ) {
59704|                        ULONG
                    | FlagValue=PSM_INDEX_FLAG_NORMAL;
59705|
59706|                        if ( Virtuallo ) {
59707|                            FlagValue =
                    | PSM_INDEX_FLAG_VIRTUAL_WRITE;

```

```

59708|                }
59709|
59710|                DataChecksum =
    | CalculateChecksum (GRANULE_SIZE, NextWritePosition);
59711|
59712|                // if virtual write,
    | save what seq. number it is for, otherwise link it to
    | the highest
59713|                // seq number found
59714|                Status = SaveIndexInfo
    | (
59715|                | DevExt,
59716|                | CurrentGranule,
59717|                | Virtuallo ? GetSequenceNumber() : SnapshotNumber,
59718|                | SavedPos,
59719|                | DataChecksum,
59720|                | FlagValue );
59721|
59722|                if ( NT_SUCCESS(Status)
    | ) {
59723|                    if ( !Virtuallo ) {
59724|                        // Mark that we
    | won't need it snapped again before another snapshot
59725|                        // NOTE It's
    | not the end of the world if we fail to credit
    | occasionally.
59726|                        // (eg no map
    | yet or asynchronous map transformation removes our
    | credit).
59727|                        // ... The
    | tree will protect us against taking a second snap for
    | the same snapshot.
59728|                        // ... The
    | worst is we might snap some free space we didn't need
    | to!!
59729|                        if (
    | Shared->Map != NULL ) {
59730|
59731| //
    | RtlClearBits ( Shared->Map, CurrentGranule, 1 );
59732|
    | ChangeTheBitsWithinBounds ( CHANGE_TO_CLEAR,
    | Shared->Map, CurrentGranule, 1 );
59733|                }

```

```

59734|                }
59735|                goto Next;
59736|            } else {
59737|                | Debug(DEBUG_DICT,("!!! Error %08x writing to index
| file\n", Status));
59738|            }
59739|        } else {
59740|            // temporary virtual
| write
59741|            #if DO_ALL_SEARCH
59742|                | Debug(DEBUG_DICT,("Skipping temporary virtual write:
| key=%016l64x\n", WorkingKey.QuadPart));
59743|            #endif
| /*DO_ALL_SEARCH*/
59744|            goto Next;
59745|        }
59746|    } else {
59747|        Debug(DEBUG_DICT,("!!!
| Error %08x writing to cache file\n", Status ));
59748|    }
59749|
59750|        // reacquire the resource so we
| can delete it from the tree
59751|        MyAcquireResourceExclusiveLite
| ( &Shared->TreeResource, TRUE );
59752|
59753|        if ( !DontFree ) {
59754|            | ASSERT(SavedPos!=INVALID_POSITION_VALUE);
59755|            // if this bit is clear,
| then we have to assume a delete has occurred
59756|            // so lets not clear it
| twice which would report the wrong
59757|            // size of the cache file.
| Also, this node could have been deleted
59758|            // so we dont access the
| node structure from this point on either.
59759|            PsmBitPositionValidate
| (DevExt->Cache.PSManBitMapBuffer, SavedPos);
59760|            if (
| RtlCheckBit(DevExt->Cache.PSManBitMapBuffer, SavedPos) )
| {
59761|                FreeCacheLocation (
| SavedPos );
59762|            }
59763|        }
59764|        if ( Status == STATUS_DISK_FULL
| ) {

```

```

59765|                Debug(DEBUG_DICT,("!!!
| pd::searchAndInsertMultiple: out of disk space:
| DevExt=%08x\n",DevExt));
59766|                Status =
| PSM_ERROR_CACHEFILE_FULL;
59767|                }
59768|            } else {
59769|                Debug(DEBUG_DICT,("!!! Error
| allocating cache file space\n" ));
59770|                Status =
| PSM_ERROR_CACHEFILE_FULL;
59771|                }
59772|                if ( !DontFree ) {
59773|                    // we delete by using the save
| key, instead of using Node->Key
59774|                    // as a race condition with a
| delete could have caused this
59775|                    // node to already be deleted.
59776|                    if ( Flags &
| DICT_FLAG_VIRTUAL_IO ) {
59777|                        Node =
| rbtree_Delete(&(Shared->VirtualWritesTree),
| WorkingKey.QuadPart);
59778|                    } else {
59779|                        Node =
| rbtree_Delete(&Shared->Tree, WorkingKey.QuadPart);
59780|                    }
59781|                } else {
59782|
| ASSERT(Node->Pos!=INVALID_POSITION_VALUE);
59783|                }
59784|            } else {
59785|                // already in tree..
59786|                ASSERT(Key.SnapShotPart>0);
59787| #if DO_ALL_IO
59788|
| Debug(DEBUG_DICT,("pd::searchAndInsertMultiple: Data
| %08I64x already in cache file\n",Key));
59789| #endif
59790|                Status = STATUS_SUCCESS;
59791|            }
59792|            if(!DontFree) {
59793|                // can be null, if something
| catastrophic happened..
59794|                if ( Node ) {
59795|                    FreeNode(Node);
59796|                }
59797|            }
59798|            MyReleaseResourceForThreadLite (
| &Shared->TreeResource );

```

```

59799|         } else {
59800|             Debug(DEBUG_DICT,( "!!! Could not
| allocate rbtree node in DasdWrite\n" ));
59801|             Status=STATUS_INSUFFICIENT_RESOURCES;
59802|         }
59803|         Next:
59804|             // update stuff for next iteration...
59805|             SectorsRemaining -= SectorsToSkip;
59806|             ++CurrentGranule;
59807|             NextWritePosition += SectorSize *
| SectorsToSkip;
59808|             SectorOffsetIntoGranule = 0;
59809|         }
59810|         Info->AlreadyHandled = TRUE;
59811|         try_exit: NOTHING;
59812|     } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
59813|         Status = GetExceptionCode();
59814|         Debug(DEBUG_DICT,("Exception %08x in
| pd::searchAndInsertMultiple\n",Status));
59815|     }
59816|
59817|     // on exit, the original write will be sent down to
| the drive
59818|
59819|
59820|     return Status;
59821| }
59822|
59823| //-----
| -----
59824|
59825|
59826| /*--- end of file perdict_search.cpp ---*/
59827|
59828|
59829|
59830| File Listing: PNP.cpp
59831|
59832| #include "precomp.h"
59833|
59834| #if _WIN32_WINNT >= 0x0500
59835|
59836| STATIC NTSTATUS
59837| PSMAN_PnpObject(
59838|     IN PDEVICE_OBJECT DeviceObject,
59839|     IN PIRP Irp
59840| );
59841| STATIC NTSTATUS
59842| PSMAN_PnpDevice(

```



```

59843|         IN PDEVICE_OBJECT DeviceObject,
59844|         IN PIRP Irp
59845|     );
59846| STATIC NTSTATUS PSMANPNPVDisk(
59847|     IN PDEVICE_OBJECT
59848|     | DeviceObject,
59849|     IN PIRP Irp
59850|     );
59851|
59852|
59853| /*-----
59854| | -----*/
59855| NTSTATUS
59856| PSMANPNP(
59857|     IN PDEVICE_OBJECT DeviceObject,
59858|     IN PIRP Irp
59859| )
59860| /*++
59861|
59862| Routine Description:
59863|
59864|     Passes the Irp to the correct handler
59865|
59866| Arguments:
59867|
59868|     DriverObject - Pointer to device object to being
59869|     | shutdown by system.
59870|     Irp          - IRP involved.
59871|
59872| Return Value:
59873|
59874|     NT Status
59875|
59876| --*/
59877| {
59878|
59879|     NTSTATUS Status;
59880|
59881|     switch ( PsmGetObjectTypes(DeviceObject) ) {
59882|     case OBJECT_INTERNAL :
59883|         Status = PSMANPNPObject(DeviceObject, Irp);
59884|         break;
59885|     case OBJECT_FILTEREDDISK :
59886|         Status = PSMANPNPDevice(DeviceObject, Irp);
59887|         break;
59888|     case OBJECT_VIRTUALDISK :
59889|         Status = PSMANPNPVDisk(DeviceObject, Irp);

```

```

59890|         break;
59891|     case OBJECT_FS_FILTER :
59892|         Status = PSMANPNPFSFilter(DeviceObject,
    | Irp);
59893|         break;
59894|     case OBJECT_FS_OBJECT :
59895|         Status = PSMANPNPFSObject(DeviceObject,
    | Irp);
59896|         break;
59897|     default:
59898|         Irp->IoStatus.Status = Status =
    | STATUS_NO_SUCH_DEVICE;
59899|         Irp->IoStatus.Information = 0 ;
59900|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
59901|         break;
59902|     }
59903|     return Status;
59904|
59905| } // end PSMANPNP()
59906|
59907|
59908| /*-----
    | -----*/
59909| STATIC NTSTATUS
59910| PSMANPNPObject(
59911|     IN PDEVICE_OBJECT DeviceObject,
59912|     IN PIRP Irp
59913| )
59914|
59915| /*++
59916|
59917| Routine Description:
59918|
59919|     This routine is called for pnp IRPs.
59920|
59921| Arguments:
59922|
59923|     DriverObject - Pointer to device object
59924|     Irp          - IRP involved.
59925|
59926| Return Value:
59927|
59928|     NT Status
59929|
59930| --*/
59931|
59932| {
59933|     NOT_REFERENCED(DeviceObject);
59934|     Debug(DEBUG_PROCCALL,("PSMANPNPObject Called\n"));
59935|     Irp->IoStatus.Status = STATUS_SUCCESS;

```

```

59936|   Irp->IoStatus.Information = 0;
59937|
59938|   IoCompleteRequest(Irp, IO_NO_INCREMENT);
59939|   Debug(DEBUG_PROCCALL,("PSManPnpObject Done\n"));
59940|   return STATUS_SUCCESS;
59941|
59942| } // end PSManPnpObject()
59943|
59944|
59945|
59946| NTSTATUS
59947| PSManRegisterDevice(
59948|     IN PDEVICE_OBJECT DeviceObject
59949| )
59950|
59951| /*++
59952|
59953| Routine Description:
59954|
59955|   Routine to initialize a proper name for the device
59956|   | object, and
59957|   register it with WMI
59958| Arguments:
59959|
59960|   DeviceObject - pointer to a device object to be
59961|   | initialized.
59962| Return Value:
59963|
59964|   Status of the initialization. NOTE: If the
59965|   | registration fails,
59966|   the device name in the DeviceExtension will be left
59967|   | as empty.
59968|
59969| --*/
59970| {
59971|     NTSTATUS          status;
59972|     IO_STATUS_BLOCK    ioStatus;
59973|     KEVENT             event;
59974|     PFILTERED_EXTENSION deviceExtension;
59975|     PIRP               irp;
59976|     STORAGE_DEVICE_NUMBER number;
59977|     ULONG               registrationFlag = 0;
59978|
59979|     PAGED_CODE();
59980|     Debug(DEBUG_PNP,("PSManRegisterDevice: DeviceObject
59981|     | %X\n",DeviceObject));

```

```

59981|    deviceExtension =
        | GetFilteredExtension(DeviceObject);
59982|    deviceExtension->IsPhysical = FALSE;
59983|    deviceExtension->DiskNumber = -1;
59984|
59985|    KeInitializeEvent(&event, NotificationEvent,
        | FALSE);
59986|
59987|    //
59988|    // Request for the device number
59989|    //
59990|    irp = IoBuildDeviceIoControlRequest(
59991|        | IOCTL_STORAGE_GET_DEVICE_NUMBER,
59992|        | deviceExtension->TargetDeviceObject,
59993|            NULL,
59994|            0,
59995|            &number,
59996|            sizeof(number),
59997|            FALSE,
59998|            &event,
59999|            &ioStatus);
60000|    if ( !irp ) {
60001|
60002|        | LogError(DeviceObject,NULL,IO_ERR_INSUFFICIENT_RESOURCES
        | ,STATUS_INSUFFICIENT_RESOURCES,NULL,0,NULL,0);
60003|        Debug(DEBUG_PNP,("PSManRegisterDevice: Fail to
        | build irp\n"));
60004|        return STATUS_INSUFFICIENT_RESOURCES;
60005|    }
60006|
60007|    status =
        | IoCallDriver(deviceExtension->TargetDeviceObject, irp);
60008|    if ( status == STATUS_PENDING ) {
60009|        pmWaitForSingleObject(&event, NULL);
60010|        status = ioStatus.Status;
60011|    }
60012|
60013|    if ( NT_SUCCESS(status) ) {
60014|
60015|        //
60016|        // Remember the disk number for use as
        | parameter in DiskIoNotifyRoutine
60017|        //
60018|
60019|        deviceExtension->DiskNumber =
        | number.DeviceNumber;
60020|

```

```

60021|    //
60022|    // Create device name for each partition
60023|    //
60024|
60025|    | swprintf(deviceExtension->PhysicalDeviceNameBuffer,L"\\D
    | evice\\Harddisk%d\\Partition%d",number.DeviceNumber,
    | number.PartitionNumber);
60026|    RtlInitUnicodeString(
    | &deviceExtension->PhysicalDeviceName,
    | &deviceExtension->PhysicalDeviceNameBuffer[0]);
60027|
60028|    | swprintf(deviceExtension->Name,L"\\Device\\Harddisk%d\\P
    | artition%d",number.DeviceNumber,
    | number.PartitionNumber);
60029|    RtlInitUnicodeString(
    | &deviceExtension->UniName, deviceExtension->Name);
60030|
60031|    if ( number.PartitionNumber == 0 ) {
60032|    //        registrationFlag =
    | WMIREG_FLAG_TRACE_PROVIDER | WMIREG_NOTIFY_DISK_IO;
60033|        deviceExtension->IsPhysical = TRUE;
60034|    }
60035|    //
60036|    // Set default name for physical disk
60037|    //
60038|    RtlCopyMemory(
    | &(deviceExtension->StorageManagerName[0]), L"PhysDisk",
    | 8 * sizeof(WCHAR));
60039|    Debug(DEBUG_PNP,("PSManRegisterDevice: Device
    | name %ws\\n",
    | deviceExtension->PhysicalDeviceNameBuffer));
60040|    } else {
60041|
60042|    // request for partition's information failed,
    | try volume
60043|
60044|    ULONG        outputSize =
    | sizeof(MOUNTDEV_NAME);
60045|    PMOUNTDEV_NAME output = (PMOUNTDEV_NAME)
    | MemAllocatePoolWithTag(PagedPool, outputSize,TEMPTAG);
60046|    if ( !output ) {
60047|
    | LogError(DeviceObject,NULL,IO_ERR_INSUFFICIENT_RESOURCES
    | ,STATUS_INSUFFICIENT_RESOURCES,NULL,0,NULL,0);
60048|        return STATUS_INSUFFICIENT_RESOURCES;
60049|    }
60050|
60051|    KeInitializeEvent(&event, NotificationEvent,

```

```

    | FALSE);
60052|     irp = IoBuildDeviceIoControlRequest(
60053|
    | IOCTL_MOUNTDEV_QUERY_DEVICE_NAME,
60054|
    | deviceExtension->TargetDeviceObject, NULL, 0,
60055|         output,
    | outputSize, FALSE, &event, &ioStatus);
60056|     if ( !irp ) {
60057|         MemFreePool(output);
60058|
    | LogError(DeviceObject,NULL,IO_ERR_INSUFFICIENT_RESOURCES
    | ,STATUS_INSUFFICIENT_RESOURCES,NULL,0,NULL,0);
60059|     return STATUS_INSUFFICIENT_RESOURCES;
60060|     }
60061|
60062|     status =
    | IoCallDriver(deviceExtension->TargetDeviceObject, irp);
60063|     if ( status == STATUS_PENDING ) {
60064|         pmWaitForSingleObject(&event, NULL);
60065|         status = ioStatus.Status;
60066|     }
60067|
60068|     if ( status == STATUS_BUFFER_OVERFLOW ) {
60069|         outputSize = sizeof(MOUNTDEV_NAME) +
    | output->NameLength;
60070|         MemFreePool(output);
60071|         output = (PMOUNTDEV_NAME)
    | MemAllocatePoolWithTag(PagedPool, outputSize,TEMPTAG);
60072|
60073|         if ( !output ) {
60074|
    | LogError(DeviceObject,NULL,IO_ERR_INSUFFICIENT_RESOURCES
    | ,STATUS_INSUFFICIENT_RESOURCES,NULL,0,NULL,0);
60075|         return STATUS_INSUFFICIENT_RESOURCES;
60076|     }
60077|
60078|     KeInitializeEvent(&event,
    | NotificationEvent, FALSE);
60079|     irp = IoBuildDeviceIoControlRequest(
60080|
    | IOCTL_MOUNTDEV_QUERY_DEVICE_NAME,
60081|
    | deviceExtension->TargetDeviceObject, NULL, 0,
60082|         output,
    | outputSize, FALSE, &event, &ioStatus);
60083|     if ( !irp ) {
60084|         MemFreePool(output);
60085|
    | LogError(DeviceObject,NULL,IO_ERR_INSUFFICIENT_RESOURCES

```

```

        | ,STATUS_INSUFFICIENT_RESOURCES,NULL,0,NULL,0);
60086|         return STATUS_INSUFFICIENT_RESOURCES;
60087|     }
60088|
60089|     status =
        | IoCallDriver(deviceExtension->TargetDeviceObject, irp);
60090|     if ( status == STATUS_PENDING ) {
60091|         pmWaitForSingleObject(&event,NULL);
60092|         status = ioStatus.Status;
60093|     }
60094| }
60095| if ( !NT_SUCCESS(status) ) {
60096|     MemFreePool(output);
60097|
        | //LogError(DeviceObject,NULL,IO_ERR_CONFIGURATION_ERROR,
        | status,NULL,0,NULL,0);
60098|     return status;
60099| }
60100|
60101|     deviceExtension->PhysicalDeviceName.Length =
        | output->NameLength;
60102|
        | deviceExtension->PhysicalDeviceName.MaximumLength =
        | output->NameLength + sizeof(WCHAR);
60103|
60104|     RtlCopyMemory(
        | deviceExtension->PhysicalDeviceName.Buffer,
        | output->Name, output->NameLength);
60105|
        | deviceExtension->PhysicalDeviceName.Buffer[deviceExtensi
        | on->PhysicalDeviceName.Length/sizeof(WCHAR)] = 0;
60106|     MemFreePool(output);
60107|
60108|
        | wcsncpy(deviceExtension->Name,deviceExtension->PhysicalDe
        | viceNameBuffer);
60109|
        | RtlInitUnicodeString(&deviceExtension->UniName,deviceExt
        | ension->Name);
60110|
60111|     RtlCopyMemory(
        | &deviceExtension->StorageManagerName[0], L"LogiDisk", 8
        | * sizeof(WCHAR));
60112|     Debug(DEBUG_PNP,("PSManRegisterDevice: Device
        | name %ws\n",
        | deviceExtension->PhysicalDeviceNameBuffer));
60113| }
60114|
60115|     status = IoWMIRegistrationControl(DeviceObject,
        | WMIREG_ACTION_REGISTER | registrationFlag );

```

```

60116|   if ( ! NT_SUCCESS(status) ) {
60117|
60118|       | LogError(DeviceObject,NULL,IO_ERR_INTERNAL_ERROR,STATUS_
60119|       | INSUFFICIENT_RESOURCES,NULL,0,NULL,0);
60120|   }
60121|   return status;
60122| }
60123| NTSTATUS
60124| PSMAN_StartDevice(
60125|     IN PDEVICE_OBJECT DeviceObject,
60126|     IN PIRP Irp
60127| )
60128| /*++
60129| Routine Description:
60130|
60131| This routine is called when a Pnp Start Irp is
60132| received.
60133| It will schedule a completion routine to initialize
60134| and register with WMI.
60135|
60136| Arguments:
60137|
60138| DeviceObject - a pointer to the device object
60139| Irp - a pointer to the irp
60140|
60141| Return Value:
60142|
60143| Status of processing the Start Irp
60144|
60145| --*/
60146| {
60147|     PFILTERED_EXTENSION deviceExtension;
60148|     NTSTATUS status;
60149|     PDISK_GEOMETRY Geometry=NULL;
60150|
60151|     PAGED_CODE();
60152|
60153|     Debug(DEBUG_PNP,("PSMAN_StartDevice: DeviceObject
60154|     | %X\n",DeviceObject));
60155|     deviceExtension =
60156|         GetFilteredExtension(DeviceObject);
60157|
60158|     status = PSManForwardIrpSynchronous(DeviceObject,
60159|         Irp);
60160|
60161|     PSManSyncFilterWithTarget(DeviceObject,

```



```

    | deviceExtension->TargetDeviceObject);
60159|
60160|    //
60161|    // Complete WMI registration
60162|    //
60163|    PSMANRegisterDevice(DeviceObject);
60164|
60165|    if ( NT_SUCCESS(status) ) {
60166|
60167|
60168|        // we can now access the lower level driver
60169|        // and get information we need.
60170|
60171|        // Allocate buffer for disk geometry.
60172|
60173|        Debug(DEBUG_INIT,("Getting geometry\n"));
60174|        Geometry =
        | (PDISK_GEOMETRY)MemAllocatePoolWithTag(PagedPool,sizeof(
        | DISK_GEOMETRY),TEMPTAG);
60175|
60176|        if ( Geometry ) {
60177|
60178|            status =
        | Sblo_GetGeometry(deviceExtension->TargetDeviceObject,Geo
        | metry);
60179|
60180|            if ( !NT_SUCCESS(status) ) {
60181|                Debug(DEBUG_INIT,("Error! %08x sending
        | disk_geometry to %X
        | '%S'\n",status,DeviceObject,deviceExtension->Name));
60182|                // keep going, incase there is no
        | cartridge in drive.
60183|                Geometry->Cylinders.QuadPart = 0;
60184|                Geometry->MediaType          =
        | RemovableMedia;
60185|                Geometry->TracksPerCylinder = 0;
60186|                Geometry->SectorsPerTrack  = 0;
60187|                Geometry->BytesPerSector   = 512;
60188|                status = 0;
60189|            }
60190|
60191|            // get the disk geometry
60192|            deviceExtension->Cylinders      =
        | Geometry->Cylinders;
60193|            deviceExtension->MediaType      =
        | Geometry->MediaType;
60194|            deviceExtension->TracksPerCylinder =
        | Geometry->TracksPerCylinder;
60195|            deviceExtension->SectorsPerTrack =
        | Geometry->SectorsPerTrack;

```

```

60196|         deviceExtension->BytesPerSector      =
        | Geometry->BytesPerSector;
60197|         deviceExtension->DeviceShutDown = 0;
60198|
60199|         // save largest BPS request
60200|         if ( deviceExtension->BytesPerSector >
        | GlobalData->LargestBPS ) {
60201|             GlobalData->LargestBPS =
        | deviceExtension->BytesPerSector;
60202|         }
60203|
60204|
60205|         Debug(DEBUG_INIT,("Device=%p, Cyls=%d,
        | Heads=%d, SPT=%d, BPS=%d\n",
60206|             DeviceObject,
60207|             | Geometry->Cylinders.LowPart,
60208|             | Geometry->TracksPerCylinder,
60209|             | Geometry->SectorsPerTrack,
60210|             Geometry->BytesPerSector
60211|             ));
60212|
60213|         FREE_POINTER(Geometry);
60214|     } else {
60215|         Debug(DEBUG_INIT,("Error! Out of
        | memory\n"));
60216|     }
60217| } else {
60218|     Debug(DEBUG_INIT,("Error %08x starting lower
        | level driver\n",status));
60219| }
60220|
60221|
60222|
60223| //
60224| // Complete the Irp
60225| //
60226| Irp->IoStatus.Status = status;
60227| IoCompleteRequest(Irp, IO_NO_INCREMENT);
60228|
60229| return status;
60230| }
60231|
60232|
60233| NTSTATUS
60234| PSMANRemoveDevice(
60235|     IN PDEVICE_OBJECT DeviceObject,
60236|     IN PIRP Irp

```

```

60237|         )
60238| /*++
60239|
60240| Routine Description:
60241|
60242|     This routine is called when the device is to be
        | removed.
60243|     It will de-register itself from WMI first, detach
        | itself from the
60244|     stack before deleting itself.
60245|
60246| Arguments:
60247|
60248|     DeviceObject - a pointer to the device object
60249|
60250|     Irp - a pointer to the irp
60251|
60252|
60253| Return Value:
60254|
60255|     Status of removing the device
60256|
60257| --*/
60258| {
60259|     NTSTATUS      status;
60260|     PFILTERED_EXTENSION deviceExtension;
60261|     PWMILIB_CONTEXT wmlibContext;
60262|
60263|     PAGED_CODE();
60264|
60265|     Debug(DEBUG_PNP,("PSManRemoveDevice: DeviceObject
        | %X
        | rc=%d\n",DeviceObject,DeviceObject->ReferenceCount));
60266|     deviceExtension =
        | GetFilteredExtension(DeviceObject);
60267|
60268|     // if psmed, then free resources for it.
60269|     if ( deviceExtension->PSMed ) {
60270|
        | PersistentDictionary::UnloadSnapShotsForVolume(
        | DeviceObject, FALSE );
60271|     }
60272|
60273|     //
60274|     // Remove registration with WMI first
60275|     //
60276|     IoWMIRegistrationControl(DeviceObject,
        | WMIREG_ACTION_DEREGISTER);
60277|
60278|     //

```

```

60279| // quickly zero out the count first to invalid the
      | structure
60280| //
60281|  WmilibContext = &deviceExtension->WmilibContext;
60282|  InterlockedExchange((PLONG)
      | &(WmilibContext->GuidCount),(LONG) 0);
60283|  RtlZeroMemory(WmilibContext,
      | sizeof(WMILIB_CONTEXT));
60284|
60285|  status = PSManForwardIrpSynchronous(DeviceObject,
      | Irp);
60286|
60287|
      | IoDetachDevice(deviceExtension->TargetDeviceObject);
60288|
60289|  KeEnterCriticalRegion();
60290|  pmAcquireWriterLock (
      | &deviceExtension->Cache.DirectAccessResource, TRUE );
60291|  DirectAccessFile *oldHeaderDirect =
      | deviceExtension->Cache.HeaderFile.Direct;
60292|  DirectAccessFile *oldIndexDirect =
      | deviceExtension->Cache.IndexFile.Direct;
60293|  DirectAccessFile *oldCacheDirect =
      | deviceExtension->Cache.CacheFile.Direct;
60294|  deviceExtension->Cache.HeaderFile.Direct = NULL;
60295|  deviceExtension->Cache.IndexFile.Direct = NULL;
60296|  deviceExtension->Cache.CacheFile.Direct = NULL;
60297|  pmReleaseWriterLock (
      | &deviceExtension->Cache.DirectAccessResource );
60298|  KeLeaveCriticalRegion();
60299|
60300|  delete oldHeaderDirect;
60301|  delete oldIndexDirect;
60302|  delete oldCacheDirect;
60303|
60304|  ExDeleteResourceLite (
      | &deviceExtension->Cache.DirectAccessResource );
60305|  pmDeRegisterObject (
      | &deviceExtension->Cache.DirectAccessResource );
60306|
60307|
60308|  // per device resources that need to be freed
60309|
      | ExDeleteResourceLite(&deviceExtension->DeviceExtResource
      | );
60310|
      | pmDeRegisterObject(&deviceExtension->DeviceExtResource);
60311|
60312|
60313|  // NOTE: Do not access deviceExtension after

```

```

    | this call as it
60314|      //      will have been freed!
60315|      IoDeleteDevice(DeviceObject);
60316|
60317|      //
60318|      // Complete the Irp
60319|      //
60320|      Irp->IoStatus.Status = status;
60321|      IoCompleteRequest(Irp, IO_NO_INCREMENT);
60322|
60323|      Debug(DEBUG_PNP,("PSManRemoveDevice: DeviceObject
    | %X done %08x\n",DeviceObject,status));
60324|      return status;
60325| }
60326|
60327|
60328|
60329| /*-----
    | -----*/
60330| STATIC NTSTATUS
60331| PSManPnpDevice(
60332|      IN PDEVICE_OBJECT DeviceObject,
60333|      IN PIRP Irp
60334|      )
60335|
60336| /*++
60337|
60338| Routine Description:
60339|
60340| This routine is called for pnp IRPs.
60341|
60342| Arguments:
60343|
60344| DriverObject - Pointer to device object
60345| Irp          - IRP involved.
60346|
60347| Return Value:
60348|
60349| NT Status
60350|
60351| --*/
60352|
60353| {
60354|      PIO_STACK_LOCATION irpSp =
    | IoGetCurrentIrpStackLocation(Irp);
60355|      NTSTATUS      status;
60356|      PFILTERED_EXTENSION
    | deviceExtension=GetFilteredExtension(DeviceObject);
60357|
60358|      PAGED_CODE();

```

```

60359|   Debug(DEBUG_PNP,("PSManPnp: Device %X Irp %X - %X -
| %s\n",DeviceObject,
| Irp,irpSp->MinorFunction,File_GetPnpMinorFunctionName(ir
| pSp->MinorFunction)));
60360|
60361|   switch ( irpSp->MinorFunction ) {
60362|
60363|       case IRP_MN_START_DEVICE: {
60364|           //
60365|           // Call the Start Routine handler to
| schedule a completion routine
60366|           //
60367|           Debug(DEBUG_PNP,("PSManPnp: Schedule
| completion for START_DEVICE\n"));
60368|           status = PSManStartDevice(DeviceObject,
| Irp);
60369|           break;
60370|
60371|       }
60372|       case IRP_MN_SURPRISE_REMOVAL : {
60373|           // if psmed, then free resources for it.
60374|           if ( deviceExtension->PSMed ) {
60375|
| PersistentDictionary::UnloadSnapShotsForVolume(
| DeviceObject, FALSE );
60376|           }
60377|           IoSkipCurrentIrpStackLocation( Irp );
60378|           return
| IoCallDriver(deviceExtension->TargetDeviceObject, Irp);
60379|       }
60380|       case IRP_MN_REMOVE_DEVICE: {
60381|           //
60382|           // Call the Remove Routine handler to
| schedule a completion routine
60383|           //
60384|           Debug(DEBUG_PNP,("PSManPnp: Schedule
| completion for REMOVE_DEVICE\n"));
60385|           status =
| PSManRemoveDevice(DeviceObject, Irp);
60386|           break;
60387|       }
60388|       case IRP_MN_DEVICE_USAGE_NOTIFICATION: {
60389|           PIO_STACK_LOCATION irpStack;
60390|           BOOLEAN setPagable;
60391|
60392|           Debug(DEBUG_PNP,("PSManPnp: Processing
| DEVICE_USAGE_NOTIFICATION\n"));
60393|           irpStack =
| IoGetCurrentIrpStackLocation(Irp);
60394|

```

```

60395|         if (
        | irpStack->Parameters.UsageNotification.Type !=
        | DeviceUsageTypePaging ) {
60396|             IoSkipCurrentIrpStackLocation( Irp
        | );
60397|             return
        | IoCallDriver(deviceExtension->TargetDeviceObject, Irp);
60398|         }
60399|
60400|         //
60401|         // wait on the paging path event
60402|         //
60403|
60404|         status =
        | pmWaitForSingleObject(&deviceExtension->PagingPathCountE
        | vent,NULL);
60405|
60406|         //
60407|         // if removing last paging device, need
        | to set DO_POWER_PAGABLE
60408|         // bit here, and possible re-set it
        | below on failure.
60409|         //
60410|
60411|         setPagable = FALSE;
60412|         if (
        | IrpStack->Parameters.UsageNotification.InPath &&
60413|         deviceExtension->PagingPathCount
        | == 1 ) {
60414|
60415|         //
60416|         // removing the last paging file
60417|         // must have DO_POWER_PAGABLE bits
        | set
60418|         //
60419|
60420|         if ( DeviceObject->Flags &
        | DO_POWER_INRUSH ) {
60421|             Debug(DEBUG_PNP,("PSManPnp:
        | last paging file removed but DO_POWER_INRUSH set, so
        | not setting PAGABLE bit for DO %p\n", DeviceObject));
60422|         } else {
60423|             Debug(DEBUG_PNP,("PSManPnp:
        | Setting PAGABLE bit for DO %p\n", DeviceObject));
60424|             DeviceObject->Flags |=
        | DO_POWER_PAGABLE;
60425|             setPagable = TRUE;
60426|         }
60427|     }
60428|

```

```

60429|          //
60430|          // send the irp synchronously
60431|          //
60432|
60433|          status =
        | PSMANForwardIrpSynchronous(DeviceObject, Irp);
60434|
60435|          //
60436|          // now deal with the failure and
        | success cases.
60437|          // note that we are not allowed to fail
        | the irp
60438|          // once it is sent to the lower
        | drivers.
60439|          //
60440|
60441|          if ( NT_SUCCESS(status) ) {
60442|
60443|          | IoAdjustPagingPathCount((PLONG)&deviceExtension->PagingP
        | athCount,irpStack->Parameters.UsageNotification.InPath);
60444|
60445|          if (
        | irpStack->Parameters.UsageNotification.InPath ) {
60446|          if (
        | deviceExtension->PagingPathCount == 1 ) {
60447|
60448|          //
60449|          // first paging file
        | addition
60450|          //
60451|
60452|          Debug(DEBUG_PNP,("PSManPnp:
        | Clearing PAGABLE bit for DO %p\n", DeviceObject));
60453|          DeviceObject->Flags &=
        | ~DO_POWER_PAGABLE;
60454|          }
60455|          }
60456|
60457|          } else {
60458|
60459|          //
60460|          // cleanup the changes done above
60461|          //
60462|
60463|          if ( setPagable == TRUE ) {
60464|          DeviceObject->Flags &=
        | ~DO_POWER_PAGABLE;
60465|          setPagable = FALSE;
60466|          }

```



```

60467|         }
60468|
60469|         //
60470|         // set the event so the next one can
        | occur.
60471|         //
60472|
60473|         | pmSetEvent(&deviceExtension->PagingPathCountEvent);
60474|
60475|         //
60476|         // and complete the irp
60477|         //
60478|
60479|         IoCompleteRequest(Irp,
        | IO_NO_INCREMENT);
60480|         return status;
60481|     }
60482|     case IRP_MN_QUERY_DEVICE_RELATIONS: {
60483|         PIO_STACK_LOCATION irpStack =
        | IoGetCurrentIrpStackLocation(Irp);
60484|
60485|         Debug(DEBUG_PNP,("PSManPnp: Processing
        | IRP_MN_QUERY_DEVICE_RELATIONS
        | %08x\n",irpStack->Parameters.QueryDeviceRelations.Type))
        | ;
60486|
60487|         IoSkipCurrentIrpStackLocation( Irp );
60488|         return
        | IoCallDriver(deviceExtension->TargetDeviceObject, Irp);
60489|     }
60490|
60491|     default:
60492|         Debug(DEBUG_PNP,("PSManPnp: Forwarding
        | irp\n"));
60493|         //
60494|         // Simply forward all other Irps
60495|         //
60496|         IoSkipCurrentIrpStackLocation( Irp );
60497|         return
        | IoCallDriver(deviceExtension->TargetDeviceObject, Irp);
60498|
60499|     }
60500|
60501|     return status;
60502|
60503| } // end PSManPnpDevice()
60504|
60505|
60506| /*-----

```

```

| -----*/
60507| STATIC NTSTATUS PSMANPNPVDisk(
60508|             IN PDEVICE_OBJECT
        | DeviceObject,
60509|             IN PIRP Irp
60510|             )
60511| {
60512|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
60513|
60514|     Debug(DEBUG_PROCCALL | DEBUG_PNP,("PSMANPNPVDisk
        | Called Dev=%p, Irp=%p\n",DeviceObject,Irp));
60515|     Irp->IoStatus.Information = 0;
60516|     Irp->IoStatus.Status = Status;
60517|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
60518|     Debug(DEBUG_PROCCALL | DEBUG_PNP,("PSMANPNPVDisk
        | Done\n"));
60519|
60520|     return Status;
60521| }
60522|
60523| /*-----
| -----*/
60524| STATIC NTSTATUS PSMANPNPFSObject(
60525|             IN PDEVICE_OBJECT
        | DeviceObject,
60526|             IN PIRP Irp
60527|             )
60528| {
60529|     NTSTATUS Status=STATUS_SUCCESS;
60530|
60531|     Debug(DEBUG_PROCCALL | DEBUG_PNP,("PSMANPNPFSObject
        | Called Dev=%p, Irp=%p\n",DeviceObject,Irp));
60532|     Irp->IoStatus.Information = 0;
60533|     Irp->IoStatus.Status = Status;
60534|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
60535|     Debug(DEBUG_PROCCALL | DEBUG_PNP,("PSMANPNPFSObject
        | Done\n"));
60536|
60537|     return Status;
60538| }
60539|
60540|
60541| /*-----
| -----*/
60542| STATIC NTSTATUS
60543| PSMANPNPFSFilter(
60544|             IN PDEVICE_OBJECT DeviceObject,
60545|             IN PIRP Irp
60546|             )
60547|

```

```

60548| /*++
60549|
60550| Routine Description:
60551|
60552|     Pass irp to handler
60553|
60554| Arguments:
60555|
60556|     DriverObject - Pointer to device object to being
        | shutdown by system.
60557|     Irp         - IRP involved.
60558|
60559| Return Value:
60560|
60561|     NT Status
60562|
60563| --*/
60564|
60565| {
60566|     NTSTATUS Status;
60567|
60568| #ifdef DEBUG
60569|     if ( PsmActive ) {
60570|         Debug(DEBUG_PNP |
        | DEBUG_PROCCALL,("PSManPnpFSFilter Called Device=%p,
        | Irp=%p\n",DeviceObject,Irp));
60571|     }
60572| #endif
60573|
60574|     Status = PSManFSPassThru( DeviceObject, Irp );
60575|
60576| #ifdef DEBUG
60577|     if ( PsmActive ) {
60578|         Debug(DEBUG_PNP |
        | DEBUG_PROCCALL,("PSManPnpFSFilter Done Device=%p,
        | Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
60579|     }
60580| #endif
60581|     return Status;
60582| } // end PSManPnpFSFilter()
60583|
60584|
60585| #endif
60586|
60587|
60588|
60589| File Listing: PNP.h
60590|
60591| NTSTATUS
60592| PSManPnp(

```

```

60593|    IN PDEVICE_OBJECT DeviceObject,
60594|    IN PIRP Irp
60595|    );
60596| NTSTATUS
60597| PSMANPnpFSObject(
60598|    IN PDEVICE_OBJECT DeviceObject,
60599|    IN PIRP Irp
60600|    );
60601| NTSTATUS
60602| PSMANPnpFSFilter(
60603|    IN PDEVICE_OBJECT DeviceObject,
60604|    IN PIRP Irp
60605|    );
60606|
60607|
60608|
60609| File Listing: POWER.cpp
60610|
60611| #include "precomp.h"
60612|
60613| #if _WIN32_WINNT >= 0x0500
60614|
60615| STATIC NTSTATUS
60616| PSMANPowerObject(
60617|    IN PDEVICE_OBJECT DeviceObject,
60618|    IN PIRP Irp
60619|    );
60620| STATIC NTSTATUS
60621| PSMANPowerDevice(
60622|    IN PDEVICE_OBJECT DeviceObject,
60623|    IN PIRP Irp
60624|    );
60625| STATIC NTSTATUS PSMANPowerVDisk(
60626|    IN PDEVICE_OBJECT DeviceObject,
60627|    IN PIRP Irp
60628|    );
60629|
60630|
60631| /*-----
    | -----*/
60632| NTSTATUS
60633| PSMANPower(
60634|    IN PDEVICE_OBJECT DeviceObject,
60635|    IN PIRP Irp
60636|    )
60637|
60638| /*++
60639|
60640| Routine Description:
60641|

```

```

60642| Passes the Irp to the correct handler
60643|
60644| Arguments:
60645|
60646| DriverObject - Pointer to device object to being
        | shutdown by system.
60647| Irp - IRP involved.
60648|
60649| Return Value:
60650|
60651| NT Status
60652|
60653| --*/
60654|
60655| {
60656|
60657| NTSTATUS Status;
60658|
60659| switch(PsmGetObjectType(DeviceObject)) {
60660|     case OBJECT_INTERNAL :
60661|         Status = PSMANPowerObject(DeviceObject,
        | Irp);
60662|         break;
60663|     case OBJECT_FILTEREDDISK :
60664|         Status = PSMANPowerDevice(DeviceObject,
        | Irp);
60665|         break;
60666|     case OBJECT_VIRTUALDISK :
60667|         Status = PSMANPowerVDisk(DeviceObject,
        | Irp);
60668|         break;
60669|     case OBJECT_FS_FILTER :
60670|         Status = PSMANPowerFSFilter(DeviceObject,
        | Irp);
60671|         break;
60672|     case OBJECT_FS_OBJECT :
60673|         Status = PSMANPowerFSObject(DeviceObject,
        | Irp);
60674|         break;
60675|     default:
60676|         Irp->IoStatus.Status = Status =
        | STATUS_NO_SUCH_DEVICE;
60677|         Irp->IoStatus.Information = 0 ;
60678|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
60679|         break;
60680| }
60681| return Status;
60682|
60683| } // end PSMANPower()
60684|

```

```

60685|
60686| /*-----
| -----*/
60687| STATIC NTSTATUS
60688| PManPowerObject(
60689|     IN PDEVICE_OBJECT DeviceObject,
60690|     IN PIRP Irp
60691| )
60692|
60693| /*++
60694|
60695| Routine Description:
60696|
60697|     This routine is called for pnp IRPs.
60698|
60699| Arguments:
60700|
60701|     DriverObject - Pointer to device object
60702|     Irp          - IRP involved.
60703|
60704| Return Value:
60705|
60706|     NT Status
60707|
60708| --*/
60709|
60710| {
60711|     NOT_REFERENCED(DeviceObject);
60712|     Debug(DEBUG_PROCCALL,("PManPowerObject
| Called\n"));
60713|     Irp->IoStatus.Status = STATUS_SUCCESS;
60714|     Irp->IoStatus.Information = 0;
60715|
60716|     IoCompleteRequest(Irp, IO_NO_INCREMENT);
60717|     Debug(DEBUG_PROCCALL,("PManPowerObject Done\n"));
60718|     return STATUS_SUCCESS;
60719|
60720| } // end PManPowerObject()
60721|
60722|
60723| /*-----
| -----*/
60724| STATIC NTSTATUS
60725| PManPowerDevice(
60726|     IN PDEVICE_OBJECT DeviceObject,
60727|     IN PIRP Irp
60728| )
60729|
60730| /*++
60731|

```

```

60732| Routine Description:
60733|
60734|   This routine is called for pnp IRPs.
60735|
60736| Arguments:
60737|
60738|   DriverObject - Pointer to device object
60739|   Irp          - IRP involved.
60740|
60741| Return Value:
60742|
60743|   NT Status
60744|
60745| --*/
60746|
60747| {
60748|   PFILTERED_EXTENSION deviceExtension =
        | GetFilteredExtension(DeviceObject);
60749|   PIO_STACK_LOCATION irpSp =
        | IoGetCurrentIrpStackLocation(Irp);
60750|
60751|   Debug(DEBUG_POWER,("PSManPowerDevice: %X %X %d -
        | %s\n", DeviceObject, Irp, irpSp->MinorFunction, File_GetPowe
        | rMinorFunctionName(irpSp->MinorFunction)));
60752|   PoStartNextPowerIrp(Irp);
60753|   IoSkipCurrentIrpStackLocation(Irp);
60754|
60755|   return
        | PoCallDriver(deviceExtension->TargetDeviceObject, Irp);
60756| } // end PSManPowerDevice()
60757|
60758|
60759| /*-----
        | -----*/
60760| STATIC NTSTATUS PSManPowerVDisk(
60761|   IN PDEVICE_OBJECT DeviceObject,
60762|   IN PIRP Irp
60763| )
60764| {
60765|   NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;;
60766|
60767|   Debug(DEBUG_PROCCALL | DEBUG_POWER,("PSManPowerVDisk
        | Called Dev=%p, Irp=%p\n", DeviceObject, Irp));
60768|   Irp->IoStatus.Information = 0;
60769|   Irp->IoStatus.Status = Status;
60770|   IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
60771|   Debug(DEBUG_PROCCALL | DEBUG_POWER,("PSManPowerVDisk
        | Done\n"));
60772|
60773|   return Status;

```

```

60774| }
60775|
60776|
60777| /*-----
    | -----*/
60778| STATIC NTSTATUS
60779| PSMANPowerFSFilter(
60780|     IN PDEVICE_OBJECT DeviceObject,
60781|     IN PIRP Irp
60782| )
60783|
60784| /*++
60785|
60786| Routine Description:
60787|
60788|     This routine is called for pnp IRPs.
60789|
60790| Arguments:
60791|
60792|     DriverObject - Pointer to device object
60793|     Irp          - IRP involved.
60794|
60795| Return Value:
60796|
60797|     NT Status
60798|
60799| --*/
60800|
60801| {
60802|     PFILTERED_EXTENSION deviceExtension =
        | GetFilteredExtension(DeviceObject);
60803|     PIO_STACK_LOCATION irpSp =
        | IoGetCurrentIrpStackLocation(Irp);
60804|
60805|     Debug(DEBUG_POWER,("PSMANPowerFSFilter: %X %X %d -
        | %s\n", DeviceObject, Irp, irpSp->MinorFunction, File_GetPowe
        | rMinorFunctionName(irpSp->MinorFunction)));
60806|     PoStartNextPowerIrp(Irp);
60807|     IoSkipCurrentIrpStackLocation(Irp);
60808|
60809|     return
        | PoCallDriver(deviceExtension->TargetDeviceObject, Irp);
60810| } // end PSMANPowerFilter()
60811|
60812|
60813| /*-----
    | -----*/
60814| STATIC NTSTATUS PSMANPowerFSObject(
60815|     IN PDEVICE_OBJECT DeviceObject,
60816|     IN PIRP Irp

```



```

60817|    )
60818| {
60819|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;;
60820|
60821|     Debug(DEBUG_PROCCALL |
        | DEBUG_POWER,("PSManPowerFSObject Called Dev=%p,
        | Irp=%p\n",DeviceObject,Irp));
60822|     Irp->IoStatus.Information = 0;
60823|     Irp->IoStatus.Status = Status;
60824|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
60825|     Debug(DEBUG_PROCCALL |
        | DEBUG_POWER,("PSManPowerFSObject Done\n"));
60826|
60827|     return Status;
60828| }
60829|
60830|
60831| #endif
60832|
60833|
60834|
60835| File Listing: POWER.h
60836|
60837| NTSTATUS
60838| PSManPower(
60839|     IN PDEVICE_OBJECT DeviceObject,
60840|     IN PIRP Irp
60841| );
60842|
60843| NTSTATUS
60844| PSManPowerFSFilter(
60845|     IN PDEVICE_OBJECT DeviceObject,
60846|     IN PIRP Irp
60847| );
60848|
60849| NTSTATUS
60850| PSManPowerFSObject(
60851|     IN PDEVICE_OBJECT DeviceObject,
60852|     IN PIRP Irp
60853| );
60854|
60855|
60856|
60857| File Listing: PRECOMP.h
60858|
60859| /*
60860| // unreferenced inline function has been removed
60861| #pragma warning (disable:4514)
60862| // unreferenced formal parameter
60863| #pragma warning (disable:4100)

```

```
60864| // conditional expression is constant
60865| #pragma warning (disable:4127)
60866|
60867| #define PSM_INCLUDE_DEAD_CODE 0
60868|
60869| //#define OSR_DEBUG
60870|
60871| // precomp.h
60872| //#define DEBUGTREE
60873| //#define DEBUGMEMORY
60874| #define NODEBUGREAD
60875| #define SYNC
60876| //#define NOPSM
60877|
60878| #ifdef _DEBUG
60879| #ifndef DEBUG
60880|     #define DEBUG
60881| #endif
60882| #endif
60883|
60884| #ifdef DEBUG
60885|     #ifndef _DEBUG
60886|         #define _DEBUG
60887|     #endif
60888|     #ifndef DBG
60889|         #define DBG 1
60890|     #endif
60891|     #define STATIC
60892| // #define OSR_DEBUG
60893|
60894|
60895| // define if you want Driver Verifier to verify the
        | Irp's we allocate
60896|     #define IRP_DEBUG
60897|
60898| // define if you want driver verifier to verify
        | memory we allocate
60899|     #define MEMDBG 0
60900|
60901| // define to use memory checking routines instead
        | of suballocating
60902|     #define _USE_MEM_CHECK_
60903|
60904| #else
60905| //     #define STATIC static
60906|     #define STATIC
60907| #endif
60908|
60909| #define TRACK_CONTENTIONS 0
60910|
```

```

60911| // define this value to print each time the functions
    | are called along with
60912| // the parameters.
60913| #define DO_ALL_IO 0
60914| #define DO_ALL_BITMAPS 0
60915| #define DO_ALL_FREESPACE 0
60916| #define DO_ALL_SEARCH 0
60917| #define DO_ALL_CLEANUP 0
60918| #define DO_ALL_SFILTER 0
60919|
60920|
60921| #ifdef LINT
60922|     // Lint doesnt like the references in ntdef.h
60923|     typedef unsigned char KIRQL;
60924|
60925|     typedef KIRQL *PKIRQL;
60926|
60927| #endif
60928|
60929| #if 0
60930| // from nt4 ntddk.h
60931| //
60932| // Logical Data Type - These are 32-bit logical values.
60933| //
60934|
60935| typedef unsigned long LOGICAL;
60936| typedef unsigned long *PLOGICAL;
60937|
60938|
60939| //
60940| // Timer type
60941| //
60942|
60943| typedef enum _TIMER_TYPE {
60944|     NotificationTimer,
60945|     SynchronizationTimer
60946| } TIMER_TYPE;
60947| #endif
60948|
60949| /*lint -emacro(641,MmGetSystemAddressForMdl)*/
60950| /*lint -emacro(742,MemAllocatePool)*/
60951| /*lint -emacro(742,MemAllocatePoolWithTag)*/
60952| /*lint -emacro(774,IoSetCompletionRoutine)*/
60953| /*lint -emacro(506,IoSetCompletionRoutine)*/
60954|
60955| extern "C"
60956| {
60957|
60958| // use ntifs.h as it has more definitions than ntddk.h
60959| /* lint -elib(10,102,763,778) */

```

```
60960|
60961| #pragma warning (push)
60962| #pragma warning (disable:4201)
60963|
60964| #ifdef OSR_DEBUG
60965| //#include <osrddk.h>
60966| #else
60967| #if _WIN32_WINNT >= 0x0500
60968| #include <ddk/ntddk.h>
60969| #else
60970| #include <ntddk.h>
60971| #endif
60972| #endif
60973| //#include "ntifs.h"
60974| #include <stdarg.h>
60975| #include <stdio.h>
60976| #include <ntdddisk.h>
60977|
60978|
60979| #if _WIN32_WINNT >= 0x0500
60980|
60981| #define INITGUID
60982| #include <ntddvol.h>
60983| #include <mountdev.h>
60984| #include "wmistr.h"
60985| #include "wmidata.h"
60986| #include "wmiguid.h"
60987| //#include "wmikm.h"
60988| #include "wmilib.h"
60989| #include "wmi.h"
60990| #include "power.h"
60991| #include "pnp.h"
60992| #undef INITGUID
60993| #include <ntddft2.h>
60994| #endif
60995|
60996| #include <ntddstor.h>
60997| #include <ntddscsi.h>
60998| #include <ntddft.h>
60999| #include <ntdddisk.h>
61000|
61001| #pragma warning (pop)
61002|
61003| /* lint -restore +elib(10,102,763,778) */
61004|
61005| // Misc Rtl functions not documented
61006| #include "rtl.h"
61007|
61008| typedef unsigned long DWORD;
61009|
```

```
61010|
61011| // public (api) header
61012| #define UNICODE
61013| #include "psm.h"
61014| #undef UNICODE
61015|
61016| // internal headers
61017| #include "irp.h"
61018| #include "rbtree.h"
61019| #include "ioctl.h"
61020| #include "bit.h"
61021| #include "mem.h"
61022|
61023| #include "ondisk.h"
61024| #include "ntfs.h"
61025|
61026| #include "reg.h"
61027| #include "file.h"
61028|
61029| //#include "cont.h"
61030| #include "primates.h"
61031| #if TRACK_CONTENTIONS
61032| #include "cont.h"
61033| #endif
61034| #include "memtrack.h"
61035| } // extern "C"
61036|
61037| // c++ header files
61038| #include "perapi.h"
61039|
61040| #include "iodirect.h"
61041| #include "virgin.h"
61042|
61043| extern "C"
61044| {
61045| // undocumented NT calls
61046| /* lint -elib(10,102,763,778)*/
61047| #include <undoc.h>
61048|
61049| #include "snapshot.h"
61050| #include "sbpsman.h"
61051| #include "devsup.h"
61052| #include "thread.h"
61053|
61054| #include "passthru.h"
61055| #include "read.h"
61056| #include "write.h"
61057| #include "create.h"
61058| #include "cleanup.h"
61059| #include "close.h"
```

```

61060| #include "flush.h"
61061| #include "shutdown.h"
61062| #include "devcon.h"
61063| #include "dcpsm.h"
61064| #include "misc.h"
61065| #include "vdisk.h"
61066| #include "log.h"
61067| #include "psmerr.h"
61068| #include "unload.h"
61069| #include "security.h"
61070| #include "sfilter.h"
61071|
61072|
61073| // new features added to sp4 that we can use
61074| #include "sp4.h"
61075| /* lint -restore +elib(10,102,763,778)*/
61076|
61077| } // extern "C"
61078|
61079| #include "revert.h"
61080|
61081| extern "C"
61082| NTSYSAPI
61083| NTSTATUS
61084| NTAPI
61085| ZwWaitForSingleObject(
61086|     IN HANDLE Handle,
61087|     IN BOOLEAN Alertable,
61088|     IN PLARGE_INTEGER Timeout OPTIONAL );
61089|
61090| #ifdef _NTIFS_
61091| // from ntddk.h that is not in ntifs.h
61092| NTHALAPI
61093| NTSTATUS
61094| IoReadPartitionTable(
61095|     IN PDEVICE_OBJECT DeviceObject,
61096|     IN ULONG SectorSize,
61097|     IN BOOLEAN ReturnRecognizedPartitions,
61098|     OUT struct _DRIVE_LAYOUT_INFORMATION
        | **PartitionBuffer
61099|     );
61100| #endif
61101|
61102| // enable this to disable exception handling, so the
        | debugger breaks where the
61103| // exception occurs
61104| #if 0
61105|     #define __try if(1)
61106|     #define _try if(1)
61107|     #define __finally if(1)

```

```

61108| #define _finally if(1)
61109| #define __except(x) else
61110| #define _except(x) else
61111| #undef GetExceptionCode
61112| #define GetExceptionCode()
    | STATUS_UNHANDLED_EXCEPTION
61113| #undef AbnormalTermination
61114| #define AbnormalTermination (FALSE)
61115| #endif
61116|
61117|
61118|
61119| File Listing: PRIMATES.h
61120|
61121| typedef enum ePrimateObjectType {
61122|     pmSpinLock,
61123|     pmMutex,
61124|     pmRwLock,
61125|     pmSemaphore,
61126|     pmEvent
61127| } tPrimateObjectType;
61128|
61129|
61130| #define pmAcquireMutex(m,t)
    | ExAcquireFastMutex(m)
61131| #define pmAcquireReaderLock(r,w)
    | ExAcquireResourceSharedLite(r,w)
61132| #define pmAcquireSpinLock(s,i)
    | KeAcquireSpinLock(s,i)
61133| #define pmAcquireWriterLock(r,w)
    | ExAcquireResourceExclusiveLite(r,w)
61134| #define pmBroadcastEvent(e)
    | KePulseEvent(e)
61135| #define pmClearEvent(e)
    | KeClearEvent(e)
61136| #define pmExamineSemaphore(s)
    | KeReadStateSemaphore(s)
61137| #define pmReleaseMutex(m)
    | ExReleaseFastMutex(m)
61138| #define pmReleaseReaderLock(r)
    | ExReleaseResourceForThreadLite(r,ExGetCurrentResourceThr
    | ead())
61139| #define pmReleaseSpinLock(s,i)
    | KeReleaseSpinLock(s,i)
61140| #define pmReleaseSemaphore(s)
    | KeReleaseSemaphore(s,1,1,FALSE);
61141| #define pmReleaseWriterLock(r)
    | ExReleaseResourceForThreadLite(r,ExGetCurrentResourceThr
    | ead())
61142| #define pmRwLockNumReaders(r)

```

```

    | ExGetSharedWaiterCount(r)
61143| #define pmRwLockedForWrite(r)
    | ExIsResourceAcquiredExclusiveLite(r)
61144| #define pmSetEvent(e)
    | KeSetEvent(e,(KPRIORITy)1,FALSE)
61145| #define pmSignalSemaphore(s)
    | KeReleaseSemaphore(s,1,1,FALSE)
61146| #define pmStartThread(t,a,h)
    | PsCreateSystemThread(h,THREAD_ALL_ACCESS,NULL,NULL,NULL,
    | t,a)
61147| #define pmWaitForSemaphore(s,t)
    | KeWaitForSingleObject(s,Suspended,(KPROCESSOR_MODE)Kerne
    | IMode,FALSE,t)
61148| #define pmWriterToReaderLock(r)
    | ExConvertExclusiveToSharedLite(r)
61149| #define pmRegisterObject(o,n,t)
61150| #define pmDeRegisterObject(o)
61151| #define pmAcquireSemaphore(s,t)
    | KeWaitForSingleObject(s,Suspended,(KPROCESSOR_MODE)Kerne
    | IMode,FALSE,t)
61152|
61153|
61154| #define pmThreadSwitch()      {
    | LARGE_INTEGER __T__={0}; KeDelayExecutionThread(
    | (KPROCESSOR_MODE)KernelMode, FALSE, &__T__); }
61155|
61156| #define pmWaitForMultipleObjects(o,n,t)
    | KeWaitForMultipleObjects(n,o,WaitAny,Suspended,(KPROCESS
    | OR_MODE)KernelMode,FALSE,t,NULL)
61157| #define pmWaitForSingleObject(o,t)
    | KeWaitForSingleObject(o,Suspended,(KPROCESSOR_MODE)Kerne
    | IMode,FALSE,t)
61158|
61159| // not implemented
61160| #define pmGetLastError()      Do Not Use
61161| #define pmInitThreadStructure(p) Do Not Use
61162| #define pmFinishedLoading()   Do Not Use
61163| #define pmCloseEvent(e)       Do Not Use
61164| #define pmCloseHandlesForThread(t) Do Not Use
61165| #define pmCloseMutex(m)       Do Not Use
61166| #define pmCloseRwLock(r)      Do Not Use
61167| #define pmCloseSemaphore(s)   Do Not Use
61168| #define pmCloseSpinLock(s)    Do Not Use
61169| #define pmCreateAutoEvent(n)   Do Not Use
61170| #define pmCreateManualEvent(n) Do Not Use
61171| #define pmCreateMutex(n,l)     Do Not Use
61172| #define pmCreateRwLock(n)      Do Not Use
61173| #define pmCreateSemaphore(n,m) Do Not Use
61174| #define pmCreateSpinLock(n)    Do Not Use
61175| #define pmSetLastError(e)      Do Not Use

```



```

61176| #define pmAssociateTidWithObject(o,t)  Do Not Use
61177| #define pmDeInitThreadStructure(p)      Do Not Use
61178| #define pmExamineMutex(m)                Do Not Use
61179| #define pmlsValidPointer(p)              Do Not Use
61180| #define pmlsValidObject(o,t)             Do Not Use
61181|
61182|
61183|
61184| File Listing: RBTREE.cpp
61185|
61186| /*
61187|
61188|   http://epaperpress.com/s\_man.html
61189|
61190|   Permission to reproduce portions of this document
61191|   | is given provided
61192|   | the web site listed below is referenced, and no
61193|   | additional
61194|   | restrictions apply. Source code, when part of a
61195|   | software project,
61196|   | may be used freely without reference to the author.
61197|
61198|   Thomas Niemann
61199|   thomasn@epaperpress.com
61200|   Portland, Oregon
61201|   epaperpress.com
61202|
61203|   http://members.xoom.com/thomasn/s\_man.htm
61204|   Portions Copyright Thomas Niemann thomasn@ips.net
61205|
61206| */
61207| /* red-black tree */
61208|
61209| #include "precomp.h"
61210|
61211| #ifndef RBTREE_TEST_PRINTF
61212| #define RBTREE_TEST_PRINTF printf
61213| #endif
61214|
61215| // #define _NO_TREE_STUFF_ 1
61216|
61217| // keep synced with original source
61218| #define NIL (Tree->TailLeaf)
61219| #define root (Tree->HeadLeaf)
61220| #define color Red
61221|
61222| #define complT(a,b) ((a) < (b))
61223| #define compEQ(a,b) ((a) == (b))
61224|

```

```

61223|
61224| // Debug Control definitions
61225| #ifdef DEBUG
61226|     #ifdef netware
61227|         #define BreakPoint(msg) OutputToScreen(
        | *psystemConsoleScreen, msg ); EnterDebugger()
61228|     #else
61229|         #ifdef _USERMODE
61230|             #undef DbgPrint
61231|             #undef DbgBreakPoint
61232|             #undef BreakPoint
61233|             #define DbgPrint printf
61234|             #define DbgBreakPoint()
61235|         #endif
61236|         #define BreakPoint(msg) {\
61237|             ULONG _Is=0, _ShouldBe=0, _Checksum=0;\
61238|             DbgPrint(msg);\
61239|             rbtree_Audit ( Tree, &_Is, &_ShouldBe,
        | &_Checksum);\
61240|             DbgPrint("Tree: Is=%08x, Shouldbe=%08x,
        | Checksum=%08x\n", _Is, _ShouldBe, _Checksum);\
61241|             DbgBreakPoint();\
61242|         }
61243|     #endif
61244| #else
61245|     #ifdef netware
61246|         extern Abend(char *msg);
61247|         #define BreakPoint(msg) Abend(msg)
61248|     #else
61249|         #define BreakPoint(msg)
61250|     #endif
61251| #endif
61252|
61253|
61254| // Thread Control Macros
61255| #ifdef netware
61256|     extern void *rbtree_Semaphore;
61257|
61258|     #define DATA_REQUEST CPSemaphore(rbtree_Semaphore)
61259|     #define DATA_RELEASE CVSemaphore(rbtree_Semaphore)
61260| #else
61261|     #define DATA_REQUEST
61262|     #define DATA_RELEASE
61263| #endif
61264|
61265|
61266| // module variables
61267|
61268| #ifdef RBTREE_TEST
61269|

```

```

61270| // Procedures used for tree testing only
61271|
61272| #define MAX_LEVEL 32
61273| #define GRAPH_WIDTH 60
61274|
61275| STATIC long LevelCount[MAX_LEVEL+1];
61276| STATIC long LevelMax, LevelGrand, LevelDepth;
61277|
61278| /*-----
    | -----*/
61279| STATIC void ClearLevel()
61280| {
61281|     int n;
61282|
61283|     // reset the count accumulators
61284|     for(n=0; n<MAX_LEVEL; n++) {
61285|         LevelCount[n] = 0;
61286|     };
61287|
61288|     // reset the max accumulator (for print scaling)
61289|     LevelMax = LevelGrand = LevelDepth = 0;
61290| }
61291|
61292| /*-----
    | -----*/
61293| STATIC void Level_CountBelow( LONG *Count, tTree *Tree,
    | tTreeLeaf *Node, int Level )
61294| {
61295|     int slot;
61296|
61297|     // exit on the tail
61298|     if(Node == NIL) {
61299|         return;
61300|     };
61301|
61302|     // recurse to process lower levels
61303|     Level_CountBelow( Count, Tree, Node->Left, Level+1
    | );
61304|     (*Count)++;
61305|
61306|     // accumulate level totals and maxima
61307|     slot = Level<MAX_LEVEL ? Level : MAX_LEVEL;
61308|     LevelCount[slot] ++;
61309|     LevelGrand ++;
61310|     LevelDepth = slot > LevelDepth ? slot : LevelDepth;
61311|     LevelMax = LevelCount[slot]>LevelMax ?
    | LevelCount[slot] : LevelMax;
61312|
61313|     // recurse for Right hand lower levels
61314|     Level_CountBelow( Count, Tree, Node->Right, Level+1

```

```

    | );
61315| }
61316|
61317| /*-----
    | -----*/
61318| STATIC void PrintLevelHistogram(void)
61319| {
61320|     int n, m, l, pdepth;
61321|     pdepth = LevelDepth < MAX_LEVEL ? LevelDepth:
        | MAX_LEVEL+1;
61322|     for (n=0; n<pdepth; n++) {
61323|         // once for each line
61324|         if (n!=MAX_LEVEL)
61325|             printf("%5d |%8ld ", n+1, LevelCount[n]);
61326|         else
61327|             printf(" >>%2d |%8ld ", n,
        | LevelCount[n]);
61328|         // compute graph scale
61329|         l = (int)((float)(LevelCount[n] / LevelMax *
        | GRAPH_WIDTH);
61330|         // one star for each scaled level
61331|         for (m=0; m<l ; m++)
61332|             printf("*");
61333|         // terminate the line
61334|         printf("\n");
61335|     };
61336|     printf("      -----\n");
61337|     printf("Total =%8ld\n", LevelGrand);
61338| }
61339|
61340|
61341| /*-----
    | -----*/
61342| void rbtree_GraphTree ( tTree *Tree )
61343| {
61344|     ULONG Count=0;
61345|
61346|     ClearLevel(); // reset accumulators
61347|     Level_CountBelow( &Count, Tree, Tree->HeadLeaf,0 );
61348|     printf("\n"); // start on a new line
61349|     PrintLevelHistogram(); // show the man how it did
61350| }
61351|
61352| /*-----
    | -----*/
61353| void rbtree_PrintBelow( LONG *Count, tTree *Tree,
        | tTreeLeaf *Node, int Level )
61354| {
61355|     int i;
61356|     ULARGE_INTEGER Key;

```

```

61357|
61358|     if(Node == NIL) {
61359|         return;
61360|     };
61361|     rbtree_PrintBelow( Count, Tree, Node->Right,
        | Level+1 );
61362|     for(i=0;i<Level;i++) {
61363|         RBTREE_TEST_PRINTF(" ");
61364|     };
61365|     if(Node->Red) {
61366|         SetConsoleTextAttribute( GetStdHandle(
        | STD_OUTPUT_HANDLE ), FOREGROUND_RED );
61367|     } else {
61368|         SetConsoleTextAttribute( GetStdHandle(
        | STD_OUTPUT_HANDLE ), FOREGROUND_BLUE );
61369|     };
61370|
61371|     Key.QuadPart = Node->Key;
61372|     RBTREE_TEST_PRINTF("%c\n",Node->Red ? 'R' : 'B');
61373|     SetConsoleTextAttribute( GetStdHandle(
        | STD_OUTPUT_HANDLE ), FOREGROUND_RED | FOREGROUND_GREEN
        | | FOREGROUND_BLUE );
61374|     (*Count)++;
61375|     rbtree_PrintBelow( Count, Tree, Node->Left, Level+1
        | );
61376| }
61377|
61378|
61379| /*-----
        | -----*/
61380| void rbtree_DumpTree ( tTree *Tree )
61381| {
61382|     ULONG Count=0;
61383|
61384|     rbtree_PrintBelow( &Count, Tree, Tree->HeadLeaf,0
        | );
61385|     RBTREE_TEST_PRINTF("Internal Count=%12d, Tree
        | Count=%12d\n",Count,Tree->NumberNodes );
61386|
61387|     if(Count != Tree->NumberNodes) {
61388|         BreakPoint("Node count doesnt match!\n");
61389|     }
61390| }
61391|
61392| #endif //def RBTREE_TEST
61393|
61394| #ifdef DEBUG
61395|
61396| /*-----
        | -----*/

```

```

61397| STATIC void rbtree_AuditBelow( LONG *Count, tTree
      | *Tree, tTreeLeaf *Node, keyType *Total, int Level )
61398| {
61399|     if(Node == NIL) {
61400|         return;
61401|     };
61402|
61403|     rbtree_AuditBelow( Count, Tree, Node->Left, Total,
      | Level+1 );
61404|     (*Count)++;
61405|     (*Total) += Node->Key;
61406|     rbtree_AuditBelow( Count, Tree, Node->Right, Total,
      | Level+1 );
61407| }
61408|
61409| /*-----
      | -----*/
61410| void rbtree_Audit ( tTree *Tree, ULONG *Is, keyType
      | *ShouldBe, keyType *Amount )
61411| {
61412|     ULONG Count=0;
61413|     keyType Total=0;
61414|
61415|     rbtree_AuditBelow( &Count, Tree,
      | Tree->HeadLeaf,&Total,0 );
61416|     *Is = Count;
61417|
61418|     *ShouldBe = Tree->NumberNodes;
61419|     *Amount = Total;
61420| }
61421|
61422|
61423| #endif
61424|
61425| #ifdef DEBUG
61426|
61427| #define DEBUG_RBTREE_HUGE      0x40000000
61428|
61429|
61430| STATIC ULONG maxdepth;
61431| STATIC ULONG depth;
61432|
61433| /*-----
      | -----*/
61434| STATIC int checkProperty(tTree *Tree,tTreeLeaf *t) {
61435|     // property 1 - Every node is red or black.
61436|     // this is given.
61437|     // property 2 - Every leaf node is black
61438|     if( (t==NIL) && (t->Red==RED)) {
61439|         Debug(DEBUG_MISC,("Property 2: node %p Leaf

```

```

    | node not BLACK\n", t));
61440|     return 0;
61441| } else
61442| // property 3 - Every Red node has both children
    | black
61443| if (t->Red == RED) {
61444|     if (t->Left->Red != BLACK) {
61445|         Debug(DEBUG_MISC,("Property 3: node %p
    | (Key=%d) Left not BLACK\n", t,t->Key));
61446|         return 0;
61447|     }
61448|     if (t->Right->Red != BLACK) {
61449|         Debug(DEBUG_MISC,("Property 3: node %p
    | (Key=%d) Right not BLACK\n", t,t->Key));
61450|         return 0;
61451|     }
61452| } else {
61453|     depth++;
61454| }
61455| // property 4 - Every path from Tree->HeadLeaf to
    | leaf contains the same number of black nodes
61456| if (t == NIL) {
61457| #if 0
61458|     if (maxdepth == -1)
61459|         maxdepth = depth;
61460|     else {
61461|         if (depth != maxdepth) {
61462|             Debug(DEBUG_MISC,("Property 4:
    | depth=%d, maxdepth=%d\n", depth, maxdepth));
61463|             return 0;
61464|         }
61465|     }
61466| #endif
61467| } else {
61468|     if(!checkProperty(Tree,t->Left))
61469|         return 0;
61470|     if(!checkProperty(Tree,t->Right))
61471|         return 0;
61472| }
61473|
61474| // left keys should always be less than us
61475| // and right keys greater than us
61476| // duplicates are not allowed
61477| if(t!=NIL) {
61478|     if(t->Left!=NIL) {
61479|         if(t->Left->Key>=t->Key) {
61480|             Debug(DEBUG_MISC,("Property 5: Left
    | %08x is greater than or equal to %08x\n",
    | t->Left->Key,t->Key ));
61481|             return 0;

```

```

61482|     }
61483| }
61484| if(t->Right!=NIL) {
61485|     if(t->Right->Key<=t->Key) {
61486|         Debug(DEBUG_MISC,("Property 5: Left
| %08x is less than or equal to %08x\n",
| t->Right->Key,t->Key ));
61487|         return 0;
61488|     }
61489| }
61490| }
61491|
61492| if (t->Red == BLACK) depth--;
61493| return 1;
61494| }
61495|
61496| /*-----
| -----*/
61497| int rbtree_CheckProperties(tTree *Tree) {
61498|     maxdepth = -1;
61499|     depth = 0;
61500|     if(!checkProperty(Tree,Tree->HeadLeaf->Left))
61501|         return 0;
61502|     if(!checkProperty(Tree,Tree->HeadLeaf->Right))
61503|         return 0;
61504|     return 1;
61505| }
61506| #endif
61507|
61508|
61509| /*-----
| -----*/
61510| STATIC void rotateLeft(tTree *Tree, tTreeLeaf *x) {
61511|
61512|     /*****
61513|      * rotate node x to Left *
61514|      *****/
61515|
61516|     tTreeLeaf *y = x->Right;
61517|
61518|     /* establish x->Right link */
61519|     x->Right = y->Left;
61520|     if (y->Left != NIL) y->Left->Parent = x;
61521|
61522|     /* establish y->Parent link */
61523|     if (y != NIL) y->Parent = x->Parent;
61524|     if (x->Parent) {
61525|         if (x == x->Parent->Left)
61526|             x->Parent->Left = y;
61527|         else

```



```

61528|         x->Parent->Right = y;
61529|     } else {
61530|         Tree->HeadLeaf = y;
61531|     }
61532|
61533|     /* link x and y */
61534|     y->Left = x;
61535|     if (x != NIL) x->Parent = y;
61536| }
61537|
61538| /*-----
| -----*/
61539| STATIC void rotateRight(tTree *Tree, tTreeLeaf *x) {
61540|
61541|     /*-----
61542|     * rotate node x to Right *
61543|     -----*/
61544|
61545|     tTreeLeaf *y = x->Left;
61546|
61547|     /* establish x->Left link */
61548|     x->Left = y->Right;
61549|     if (y->Right != NIL) y->Right->Parent = x;
61550|
61551|     /* establish y->Parent link */
61552|     if (y != NIL) y->Parent = x->Parent;
61553|     if (x->Parent) {
61554|         if (x == x->Parent->Right)
61555|             x->Parent->Right = y;
61556|         else
61557|             x->Parent->Left = y;
61558|     } else {
61559|         Tree->HeadLeaf = y;
61560|     }
61561|
61562|     /* link x and y */
61563|     y->Right = x;
61564|     if (x != NIL) x->Parent = y;
61565| }
61566|
61567| /*-----
| -----*/
61568| STATIC void insertFixup(tTree *Tree, tTreeLeaf *x) {
61569|
61570|     /*-----
61571|     * maintain Red-Black tree balance *
61572|     * after inserting node x *
61573|     -----*/
61574|
61575|     /* check Red-Black properties */

```

```

61576|   while (x != Tree->HeadLeaf && x->Parent->Red ==
      | RED) {
61577|       /* we have a violation */
61578|       if (x->Parent == x->Parent->Parent->Left) {
61579|           tTreeLeaf *y = x->Parent->Parent->Right;
61580|           if (y->Red == RED) {
61581|
61582|               /* uncle is RED */
61583|               x->Parent->Red = BLACK;
61584|               y->Red = BLACK;
61585|               x->Parent->Parent->Red = RED;
61586|               x = x->Parent->Parent;
61587|           } else {
61588|
61589|               /* uncle is BLACK */
61590|               if (x == x->Parent->Right) {
61591|                   /* make x a Left child */
61592|                   x = x->Parent;
61593|                   rotateLeft(Tree,x);
61594|               }
61595|
61596|               /* recolor and rotate */
61597|               x->Parent->Red = BLACK;
61598|               x->Parent->Parent->Red = RED;
61599|               rotateRight(Tree,x->Parent->Parent);
61600|           }
61601|       } else {
61602|
61603|           /* mirror image of above code */
61604|           tTreeLeaf *y = x->Parent->Parent->Left;
61605|           if (y->Red == RED) {
61606|
61607|               /* uncle is RED */
61608|               x->Parent->Red = BLACK;
61609|               y->Red = BLACK;
61610|               x->Parent->Parent->Red = RED;
61611|               x = x->Parent->Parent;
61612|           } else {
61613|
61614|               /* uncle is BLACK */
61615|               if (x == x->Parent->Left) {
61616|                   x = x->Parent;
61617|                   rotateRight(Tree,x);
61618|               }
61619|               x->Parent->Red = BLACK;
61620|               x->Parent->Parent->Red = RED;
61621|               rotateLeft(Tree,x->Parent->Parent);
61622|           }
61623|       }
61624|   }

```

```

61625|   Tree->HeadLeaf->Red = BLACK;
61626| }
61627|
61628|
61629| /*-----
| -----*/
61630| int rbtree_Insert(tTree *Tree, tTreeLeaf *x ) {
61631| #ifndef _NO_TREE_STUFF_
61632|   ASSERT(++Tree->Writers==1);
61633|   ASSERT(Tree->Readers==0);
61634|
61635|   tTreeLeaf *current, *Parent;
61636| #ifdef DEBUG
61637|   tTreeLeaf *Grandpa;
61638| #endif
61639|
61640| //   Debug(DEBUG_RBTREE_HUGE,("rbtree: insert   :
| t=%08x, k=%016l64x, p=%08x\n",Tree,x->Key,x->Pos));
61641|
61642| /*****
61643|  * allocate node for data and insert in tree *
61644|  *****/
61645|
61646|   DATA_REQUEST;
61647|
61648|   /* find future Parent */
61649|   current = Tree->HeadLeaf;
61650|   Parent = 0;
61651| #ifdef DEBUG
61652|   Grandpa = 0;
61653| #endif
61654|   while (current != NIL) {
61655| #ifdef DEBUG
61656|     ASSERT((ULONG)current!=0xBAADF00D);
61657| #endif
61658|     ASSERT(current!=NULL);
61659|     if (compEQ(x->Key, current->Key)) {
61660|       DATA_RELEASE;
61661|       ASSERT(--Tree->Writers==0);
61662|       return STATUS_DUPLICATE_OBJECTID;
61663|     }
61664| #ifdef DEBUG
61665|     Grandpa = Parent;
61666| #endif
61667|     Parent = current;
61668|     current = compLT(x->Key, current->Key) ?
61669|       current->Left : current->Right;
61670|   }
61671|
61672|   /* setup new node */

```

```

61673| x->Parent = Parent;
61674| x->Left = NIL;
61675| x->Right = NIL;
61676| x->Red = RED;
61677|
61678| /* insert node in tree */
61679| if(Parent) {
61680|     if(compLT(x->Key, Parent->Key))
61681|         Parent->Left = x;
61682|     else
61683|         Parent->Right = x;
61684| } else {
61685|     Tree->HeadLeaf = x;
61686| }
61687|
61688| insertFixup(Tree,x);
61689|
61690| #ifdef DEBUG
61691|     pTreeLeaf Temp=rbtree_Search(Tree,x->Key);
61692|     ASSERT(Temp==x);
61693|     maxdepth = -1;
61694|     depth = 0;
61695|     if(Grandpa) {
61696|         ASSERT(checkProperty(Tree,Grandpa));
61697|     } else
61698|     if(Parent) {
61699|         ASSERT(checkProperty(Tree,Parent));
61700|     } else {
61701|         ASSERT(Tree->HeadLeaf == x);
61702|     }
61703| #endif
61704|
61705|     Tree->NumberNodes++;
61706|     DATA_RELEASE;
61707|     ASSERT(--Tree->Writers==0);
61708| #endif
61709|     return STATUS_SUCCESS;
61710| }
61711|
61712| /*-----
| -----*/
61713| STATIC void deleteFixup(tTree *Tree, tTreeLeaf *x) {
61714|
61715|     /*-----
61716|     * maintain Red-Black tree balance *
61717|     * after deleting node x *
61718|     -----*/
61719|
61720|     while (x != Tree->HeadLeaf && x->Red == BLACK) {
61721|         if (x == x->Parent->Left) {

```

```

61722|         tTreeLeaf *w = x->Parent->Right;
61723|         if (w->Red == RED) {
61724|             w->Red = BLACK;
61725|             x->Parent->Red = RED;
61726|             rotateLeft (Tree,x->Parent);
61727|             w = x->Parent->Right;
61728|         }
61729|         if (w->Left->Red == BLACK && w->Right->Red
| == BLACK) {
61730|             w->Red = RED;
61731|             x = x->Parent;
61732|         } else {
61733|             if (w->Right->Red == BLACK) {
61734|                 w->Left->Red = BLACK;
61735|                 w->Red = RED;
61736|                 rotateRight (Tree,w);
61737|                 w = x->Parent->Right;
61738|             }
61739|             w->Red = x->Parent->Red;
61740|             x->Parent->Red = BLACK;
61741|             w->Right->Red = BLACK;
61742|             rotateLeft (Tree,x->Parent);
61743|             x = Tree->HeadLeaf;
61744|         }
61745|     } else {
61746|         tTreeLeaf *w = x->Parent->Left;
61747|         if (w->Red == RED) {
61748|             w->Red = BLACK;
61749|             x->Parent->Red = RED;
61750|             rotateRight (Tree,x->Parent);
61751|             w = x->Parent->Left;
61752|         }
61753|         if (w->Right->Red == BLACK && w->Left->Red
| == BLACK) {
61754|             w->Red = RED;
61755|             x = x->Parent;
61756|         } else {
61757|             if (w->Left->Red == BLACK) {
61758|                 w->Right->Red = BLACK;
61759|                 w->Red = RED;
61760|                 rotateLeft (Tree,w);
61761|                 w = x->Parent->Left;
61762|             }
61763|             w->Red = x->Parent->Red;
61764|             x->Parent->Red = BLACK;
61765|             w->Left->Red = BLACK;
61766|             rotateRight (Tree,x->Parent);
61767|             x = Tree->HeadLeaf;
61768|         }
61769|     }

```

```

61770| }
61771| x->Red = BLACK;
61772| }
61773|
61774| /*-----
| -----*/
61775| tTreeLeaf *rbtree_Delete(tTree *Tree, keyType Key) {
61776| #ifndef _NO_TREE_STUFF_
61777|     ASSERT(++Tree->Writers==1);
61778|     ASSERT(Tree->Readers==0);
61779|
61780|     tTreeLeaf *x, *y, *z;
61781|
61782| //     Debug(DEBUG_RBTREE_HUGE,("rbtree: delete
| t=%08x, k=%016l64x\n", Tree, Key));
61783| /*****
61784|  * delete node z from tree  *
61785|  *****/
61786|
61787|     DATA_REQUEST;
61788|
61789|     /* find node in tree */
61790|     z = Tree->HeadLeaf;
61791|     while(z != NIL) {
61792| #ifdef DEBUG
61793|         ASSERT((ULONG)z!=0xBAADF00D);
61794| #endif
61795|         if(compEQ(Key, z->Key))
61796|             break;
61797|         else
61798|             z = compLT(Key, z->Key) ? z->Left :
| z->Right;
61799|     }
61800|     if (z==NIL) {
61801|         DATA_RELEASE;
61802|         ASSERT(--Tree->Writers==0);
61803|         return NULL;
61804|     }
61805|
61806|     if (z->Left == NIL || z->Right == NIL) {
61807|         /* y has a NIL node as a child */
61808|         y = z;
61809|     } else {
61810|         /* find tree successor with a NIL node as a
| child */
61811|         y = z->Right;
61812|         while (y->Left != NIL) y = y->Left;
61813|     }
61814|
61815|     /* x is y's only child */

```

```

61816|  if (y->Left != NIL)
61817|      x = y->Left;
61818|  else
61819|      x = y->Right;
61820|
61821|  /* remove y from the Parent chain */
61822|  x->Parent = y->Parent;
61823|  if (y->Parent)
61824|      if (y == y->Parent->Left)
61825|          y->Parent->Left = x;
61826|      else
61827|          y->Parent->Right = x;
61828|  else
61829|      Tree->HeadLeaf = x;
61830|
61831|  // we are actually going to delete the
61832|  // next lowest number, and move its data
61833|  // into our node
61834|  if (y != z) {
61835|      keyType Key=z->Key;
61836|      ULONG   Pos=z->Pos;
61837|      z->Key = y->Key;
61838|      z->Pos = y->Pos;
61839|      y->Pos = Pos;
61840|      y->Key = Key;
61841|  }
61842|
61843|
61844|  if (y->Red == BLACK)
61845|      deleteFixup (Tree,x);
61846|
61847|
61848| #ifdef DEBUG
61849|  // make sure theres not still one in the tree
61850|  y->Parent = (tTreeLeaf*)0xBAADF00D;
61851|  y->Right  = (tTreeLeaf*)0xBAADF00D;
61852|  y->Left   = (tTreeLeaf*)0xBAADF00D;
61853|  if(Tree->NumberNodes>1) {
61854|      // theres a bug somewhere in here when we
        | delete
61855|      // the last node in the tree. I dont have time
61856|      // to find it, and it only affects debug builds
61857|      // rob - 4-7-2001
61858|      ASSERT(rbtree_Search(Tree,y->Key)==NULL);
61859|  } else {
61860|      ASSERT(Tree->NumberNodes==1);
61861|  }
61862| #endif
61863|
61864|  Tree->NumberNodes--;

```

```

61865|
61866| // y may not equal the node they searched for! (z
    | is)
61867| DATA_RELEASE;
61868| ASSERT(--Tree->Writers==0);
61869|
61870| if(Tree->NumberNodes==0) {
61871|     rbtree_Init(Tree);
61872| }
61873| return y;
61874| #else
61875| return NULL;
61876| #endif
61877| }
61878|
61879| /*-----
    | -----*/
61880| tTreeLeaf *rbtree_Search(tTree *Tree, keyType Key ) {
61881| #ifndef _NO_TREE_STUFF_
61882|     ASSERT(InterlockedIncrement(&Tree->Readers)>0);
61883|
61884|     /*****
61885|      * find node containing data *
61886|      *****/
61887|
61888|     tTreeLeaf *current = Tree->HeadLeaf;
61889|
61890|     DATA_REQUEST;
61891|
61892|     while(current != NIL) {
61893| #ifdef DEBUG
61894|         ASSERT((ULONG)current!=0xBAADF00D);
61895| #endif
61896|         if(compEQ(Key, current->Key)) {
61897|             DATA_RELEASE;
61898|
        | ASSERT(InterlockedDecrement(&Tree->Readers)>=0);
61899|             return current;
61900|         } else {
61901|             current = compLT (Key, current->Key) ?
61902|                 current->Left : current->Right;
61903|         }
61904|     }
61905|     DATA_RELEASE;
61906| #endif
61907|     ASSERT(InterlockedDecrement(&Tree->Readers)>=0);
61908|     return NULL;
61909| }
61910|
61911| //-----

```



```

| -----
61912|
61913| tTreeLeaf *rbtree_GetNextInOrder( tTree *Tree,
| tTreeLeaf *Current )
61914| {
61915|     ASSERT(InterlockedIncrement(&Tree->Readers)>0);
61916|     if(Current->Right!=NIL) {
61917|         // find left most node which will be the next
| highest
61918|         Current = Current->Right;
61919|         while(Current->Left!=NIL) {
61920|             Current = Current->Left;
61921|         }
61922|     } else {
61923|         // search until
61924|         while((Current->Parent!=NULL) &&
| (Current->Parent->Right==Current)) {
61925|             Current=Current->Parent;
61926|         }
61927|         Current=Current->Parent;
61928|     }
61929|     ASSERT(InterlockedDecrement(&Tree->Readers)>=0);
61930|     return Current;
61931| }
61932|
61933| //-----
| -----
61934|
61935| tTreeLeaf *rbtree_SearchUpperBound ( tTree *Tree,
| keyType Key )
61936| {
61937|     ASSERT(InterlockedIncrement(&Tree->Readers)>0);
61938|
61939|     tTreeLeaf *upperBound = NULL;
61940|     tTreeLeaf *current = NIL;
61941|
61942|     DATA_REQUEST;
61943|
61944|     current = Tree->HeadLeaf;
61945|     while ( current != NIL ) {
61946|         if(compEQ(Key, current->Key)) {
61947|             DATA_RELEASE;
61948|
| ASSERT(InterlockedDecrement(&Tree->Readers)>=0);
61949|             return current;
61950|         } else {
61951|             if ( compLT(Key,current->Key) ) {
61952|                 upperBound = current;
61953|                 current = current->Left;
61954|             } else {

```

```

61955|         current = current->Right;
61956|     }
61957| }
61958| }
61959|
61960| DATA_RELEASE;
61961| ASSERT(InterlockedDecrement(&Tree->Readers)>=0);
61962| return upperBound;
61963| }
61964|
61965|
61966| /*-----
| -----*/
61967| void rbtree_DeleteAll( tTree *Tree, void
| (*FreeFunc)(void *Handle) )
61968| {
61969|     tTreeLeaf *Node;
61970|
61971|     ASSERT(Tree->Writers==0);
61972|     ASSERT(Tree->Readers==0);
61973| //     Debug(DEBUG_RBTREE_HUGE,("rbtree: deleteall :
| t=%08x\n",Tree));
61974|     Node=Tree->HeadLeaf;
61975|     while( Node!=NIL ) {
61976|         FreeFunc( rbtree_Delete(Tree, Node->Key) );
61977|         Node = Tree->HeadLeaf;
61978|     }
61979|     // reinit tree...
61980|     rbtree_Init( Tree );
61981| }
61982|
61983| /*-----
| -----*/
61984| void rbtree_DeleteAllQuick( tTree *Tree )
61985| {
61986|     ASSERT(Tree->Writers==0);
61987|     ASSERT(Tree->Readers==0);
61988| //     Debug(DEBUG_RBTREE_HUGE,("rbtree: deleteallq:
| t=%08x\n",Tree));
61989|     // reinit tree...
61990|     rbtree_Init( Tree );
61991| }
61992|
61993| /*-----
| -----*/
61994| void rbtree_Init( tTree *Tree ) {
61995|
61996| //     Debug(DEBUG_RBTREE_HUGE,("rbtree: init :
| t=%08x\n",Tree));
61997|     DATA_REQUEST;

```

```

61998|   Tree->TailLeaf = &Tree->TheTailStorage;
61999|   Tree->TailLeaf->Left = NIL;
62000|   Tree->TailLeaf->Right = NIL;
62001|   Tree->TailLeaf->Red = BLACK;
62002|   Tree->TailLeaf->Pos = INVALID_POSITION_VALUE;
62003|   Tree->TailLeaf->Key = 0;
62004|   Tree->HeadLeaf = NIL;
62005|   Tree->NumberNodes = 0;
62006| #ifdef DEBUG
62007|   Tree->Readers = 0;
62008|   Tree->Writers = 0;
62009| #endif
62010|   DATA_RELEASE;
62011| }
62012|
62013|
62014| /*-----
   | -----*/
62015|
62016|
62017|
62018| File Listing: RBTREE.h
62019|
62020| #define MAXTREES MAXDEVICES
62021|
62022| /*lint -save -e652*/
62023| #ifndef LONG
62024| #define LONG unsigned long
62025| #endif
62026|
62027| #ifndef ULONG
62028| #define ULONG unsigned long
62029| #endif
62030|
62031| #ifndef NULL
62032| #define NULL 0
62033| #endif
62034|
62035| #ifndef MAXDEVICES
62036| #define MAXDEVICES 32
62037| #endif
62038|
62039| #ifndef ULONGLONG
62040| #define ULONGLONG unsigned __int64
62041| #endif
62042|
62043| #ifndef BYTE
62044| #define BYTE unsigned char
62045| #endif
62046| /*lint -restore*/

```

```

62047|
62048| #ifndef STATUS_SUCCESS
62049| typedef ULONG NTSTATUS;
62050| #define STATUS_INSUFFICIENT_RESOURCES
    | ((NTSTATUS)0xC000009AL) // ntsubauth
62051| #define STATUS_DUPLICATE_OBJECTID
    | ((NTSTATUS)0xC000022AL)
62052| #define STATUS_NONEXISTENT_SECTOR
    | ((NTSTATUS)0xC0000015L)
62053| #define STATUS_KEY_NOT_FOUND
    | STATUS_NONEXISTENT_SECTOR
62054| #define STATUS_SUCCESS
    | ((NTSTATUS)0x00000000L) // ntsubauth
62055| #endif
62056|
62057| typedef ULONGLONG keyType; /* type of key */
62058|
62059| /* implementation independent declarations */
62060| /* Red-Black tree description */
62061|
62062| #define RED 1
62063| #define BLACK 0
62064|
62065| typedef struct sTreeLeaf {
62066|     struct sTreeLeaf *Left; /* left child */
62067|     struct sTreeLeaf *Right; /* right child */
62068|     struct sTreeLeaf *Parent; /* parent */
62069|     ULONG Pos:31;
62070|     ULONG Red:1;
62071|     keyType Key; /* key used for
    | searching */
62072| } tTreeLeaf,*pTreeLeaf;
62073|
62074| // tTreeLeaf->Pos is only 31 bits
62075| #define INVALID_POSITION_VALUE (0x7fffffff)
62076|
62077| typedef struct sTree {
62078|     tTreeLeaf *TailLeaf;
62079|     tTreeLeaf *HeadLeaf;
62080|     tTreeLeaf TheTailStorage;
62081|     ULONG NumberNodes;
62082| #ifdef DEBUG
62083|     signed long Writers;
62084|     signed long Readers;
62085| #endif
62086| } tTree,*pTree;
62087|
62088|
62089|
62090| void rbtree_Init( tTree *Tree ); /* must be

```

```

    | called prior to using this lib
62091| int rbtree_Insert( tTree *Tree, tTreeLeaf *Node );
    | // 0 if so inserted otherwise error code
62092| tTreeLeaf *rbtree_Search( tTree *Tree, keyType Key);
    | // returns ==TailLeaf if not found
62093| tTreeLeaf *rbtree_Delete( tTree *Tree, keyType Key);
    | // returns pointer to node deleted.
62094| void rbtree_DeleteAll( tTree *Tree, void
    | (*FreeFunc)(void *Handle) );
62095| tTreeLeaf *rbtree_SearchUpperBound ( tTree *Tree,
    | keyType Key );
62096| tTreeLeaf *rbtree_SearchLowerBound ( tTree *Tree,
    | keyType Key );
62097| tTreeLeaf *rbtree_GetNextInOrder( tTree *Tree,
    | tTreeLeaf *Current );
62098| tTreeLeaf *rbtree_GetPrevInOrder( tTree *Tree,
    | tTreeLeaf *Current );
62099|
62100|
62101| #ifdef RBTREE_TEST
62102| void rbtree_DumpTree ( tTree *Tree );
62103| int rbtree_CheckProperties(tTree *Tree);
62104|
62105| void rbtree_Count ( tTree *Tree, LONG *Is, LONG
    | *ShouldBe );
62106| void rbtree_Audit ( tTree *Tree, LONG *Is, LONG
    | *ShouldBe, LONG *Total );
62107| void rbtree_GraphTree ( tTree *Tree );
62108| #endif
62109|
62110|
62111|
62112| File Listing: READ.cpp
62113|
62114| #include "precomp.h"
62115|
62116|
62117| /*-----
    | -----*/
62118| NTSTATUS
62119| PSMANRead(
62120|     IN PDEVICE_OBJECT DeviceObject,
62121|     IN PIRP Irp
62122| )
62123|
62124| /*++
62125|
62126| Routine Description:
62127|
62128|     This is the driver entry point for read requests

```

```

62129|   to disks to which the PSMAN driver has attached.
62130|   This driver collects statistics and then sets a
62131|   | completion
62132|   routine so that it can collect additional
62133|   | information when
62134|   the request completes. Then it calls the next
62135|   | driver below
62136|   it.
62137| Arguments:
62138|   DeviceObject
62139|   Irp
62140| Return Value:
62141|   NTSTATUS
62142| --*/
62143| {
62144|   switch(PsmGetObjectTypes(DeviceObject)) {
62145|     case OBJECT_INTERNAL :
62146|       return PSMANReadObject(DeviceObject, Irp);
62147|     case OBJECT_FILTEREDDISK :
62148|       return PSMANReadDevice(DeviceObject, Irp);
62149|     case OBJECT_VIRTUALDISK :
62150|       return PSMANReadVDisk(DeviceObject, Irp);
62151|     case OBJECT_FS_FILTER :
62152|       return PSMANReadFSFilter(DeviceObject,
62153|       | Irp);
62154|     case OBJECT_FS_OBJECT :
62155|       return PSMANReadFSObject(DeviceObject,
62156|       | Irp);
62157|     default:
62158|       Irp->IoStatus.Status = STATUS_NO_SUCH_DEVICE;
62159|       Irp->IoStatus.Information = 0 ;
62160|       IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
62161|       return STATUS_NO_SUCH_DEVICE;
62162|   }
62163| } // PSMANRead
62164|
62165|
62166|
62167| /*-----
62168| | -----*/
62169| STATIC NTSTATUS
62170| PSMANReadObject(
62171|   IN PDEVICE_OBJECT DeviceObject,
62172|   IN PIRP Irp
62173| )

```

```

62173|
62174| /*++
62175|
62176| Routine Description:
62177|
62178|   This is the driver entry point for read requests
62179|   to disks to which the PSMAN driver has attached.
62180|   This driver collects statistics and then sets a
        | completion
62181|   routine so that it can collect additional
        | information when
62182|   the request completes. Then it calls the next
        | driver below
62183|   it.
62184|
62185| Arguments:
62186|
62187|   DeviceObject
62188|   Irp
62189|
62190| Return Value:
62191|
62192|   NTSTATUS
62193|
62194| --*/
62195|
62196| {
62197|   NTSTATUS Status=STATUS_INVALID_PARAMETER;
62198|   NOT_REFERENCED(DeviceObject);
62199|
62200|   Debug(DEBUG_PROCCALL,("PSManReadObject Called\n"));
62201|   Irp->IoStatus.Status = Status;
62202|   Irp->IoStatus.Information = 0;
62203|
62204|   IoCompleteRequest(Irp, IO_NO_INCREMENT);
62205|   Debug(DEBUG_PROCCALL,("PSManReadObject Done\n"));
62206|   return Status;
62207| } // PSManReadObject
62208|
62209|
62210| /*-----
        | -----*/
62211| STATIC NTSTATUS
62212| PSManReadDevice(
62213|   IN PDEVICE_OBJECT DeviceObject,
62214|   IN PIRP Irp
62215|   )
62216|
62217| /*++
62218|

```

```

62219| Routine Description:
62220|
62221|   This is the driver entry point for read requests
62222|   to disks to which the PSMAN driver has attached.
62223|   This driver collects statistics and then sets a
        | completion
62224|   routine so that it can collect additional
        | information when
62225|   the request completes. Then it calls the next
        | driver below
62226|   it.
62227|
62228| Arguments:
62229|
62230|   DeviceObject
62231|   Irp
62232|
62233| Return Value:
62234|
62235|   NTSTATUS
62236|
62237| --*/
62238|
62239| {
62240|
62241|   | ASSERT(PsmGetObjectype(DeviceObject)==OBJECT_FILTEREDDI
        | SK);
62242|   // tell psm that we have an io pending
62243|   GetGlobalDeviceForRead();
62244|   __try {
62245|
62246| #ifdef DEBUG
62247|     if(((PFILTERED_EXTENSION)
        | GetDeviceExtension(DeviceObject))->PSMed) {
62248|       PIO_STACK_LOCATION currentIrpStack =
        | IoGetCurrentIrpStackLocation(Irp);
62249|       TRACE( TRACE_READ,
62250|         0,
62251|         | (long)(currentIrpStack->Parameters.Read.ByteOffset.QuadP
        | art / 512),
62252|         currentIrpStack->Parameters.Read.Length
        | / 512,
62253|         currentIrpStack->Parameters.Read.Key,
62254|         "");
62255|     }
62256| #endif
62257| #if 0
62258|     if(DevExt->PSMed) {

```



```

62259|    // for testing bad sector handling.
62260|        if(1) {
62261|            #define NUMBADSECTORS (1)
62262|            ULONG BadSector[] = { 8370841 };
62263|
62264|            // if our thread, and the physical
        | object, inject some bad sectors
62265|            if(!Audit) &&
62266|                (DevExt->IsPhysical)) {
62267|                int i;
62268|                ULARGE_INTEGER UL;
62269|                ULONG Remainder;
62270|                ULONG Sector,Count;
62271|
62272|                // physical sector on disk...
62273|                UL.QuadPart =
        | currentIrpStack->Parameters.Read.ByteOffset.QuadPart +
62274|        | DevExt->StartingOffset.QuadPart;
62275|
62276|                Sector = RtlEnlargedUnsignedDivide
        | ( UL, DevExt->BytesPerSector, &Remainder);
62277|                Count =
        | currentIrpStack->Parameters.Read.Length /
        | DevExt->BytesPerSector;
62278|
62279|                for(i=0;i<NUMBADSECTORS;i++) {
62280|                    if( (BadSector[i] >= Sector) &&
        | (BadSector[i] < Sector+Count)) {
62281|                        // we have a sector in
        | range
62282|        | Debug(DEBUG_READ,("Returning bad sector %d in read %d
        | for %d\n",BadSector[i],Sector,Count));
62283|                Irp->IoStatus.Status =
        | STATUS_DATA_ERROR;
62284|                Irp->IoStatus.Information =
        | 0;
62285|
62286|                IoCompleteRequest(Irp,
        | IO_NO_INCREMENT);
62287|                return STATUS_DATA_ERROR;
62288|            }
62289|        }
62290|    }
62291| }
62292| }
62293|    // end of testing for bad sectors handling.
62294| #endif
62295|

```

```

62296|      // acquire physical spin lock
62297|      {
62298|          PFILTERED_EXTENSION DevExt =
        | GetFilteredExtension(DeviceObject);
62299|          PIO_STACK_LOCATION currentIrpStack =
        | IoGetCurrentIrpStackLocation(Irp);
62300|          KIRQL oldIrql;
62301|          pmAcquireSpinLock (
        | &DevExt->StatisticsSpinLock, &oldIrql );
62302|          // update the logical partition
62303|          DevExt->SectorsRead +=
        | (currentIrpStack->Parameters.Read.Length /
        | DevExt->BytesPerSector );
62304|          DevExt->NumberOfReadRequests++;
62305|
62306|          // update the physical drive
62307|          //PhyExt->SectorsRead +=
        | (currentIrpStack->Parameters.Read.Length /
        | PhyExt->BytesPerSector);
62308|          //PhyExt->NumberOfReadRequests++;
62309|          pmReleaseSpinLock(
        | &DevExt->StatisticsSpinLock, oldIrql );
62310|      }
62311|
62312|
        | if((GetFilteredExtension(DeviceObject))->SignalRead) {
62313|          // inform about read.
62314|
        | pmSetEvent(&(GetFilteredExtension(DeviceObject))->ReadEv
        | ent);
62315|      }
62316| #if 0
62317|      if(PhyExt->SignalRead) {
62318|          // inform about read.
62319|          pmSetEvent(&(PhyExt->ReadEvent));
62320|      }
62321| #endif
62322| /*
62323|      if((DevExt->PSMed) || (PhyExt->PSMed)) {
62324|          Debug(DEBUG_INFO,("Read:
        | PsGetCurrentProcess=%08x "
62325|          "PsGetCurrentThread=%08x "
62326|          "IoGetCurrentProcess=%08x "
62327|          "KeGetCurrentThread=%08x "
62328|          "User Thread=%08x\n",
62329|          PsGetCurrentProcess(),
62330|          PsGetCurrentThread(),
62331|          IoGetCurrentProcess(),
62332|          KeGetCurrentThread(),
62333|          Irp->Tail.Overlay.Thread

```

```

62334|         ));
62335|     }
62336|     */
62337|
62338| } __finally {
62339|     ReleaseGlobalDeviceForRead();
62340| }
62341|
62342| return PSManPassThru(DeviceObject,Irp);
62343|
62344| } // PSManReadDevice
62345|
62346| /*-----
| -----*/
62347| STATIC NTSTATUS ReadPreProcessSpecialSectors (
| PDEVICE_OBJECT DeviceObject,
62348|         ULONG
| LogSector,
62349|         ULONG
| Count,
62350|         char
| *Buffer )
62351| {
62352|     NOT_REFERENCED(DeviceObject);
62353|     NOT_REFERENCED(LogSector);
62354|     NOT_REFERENCED(Count);
62355|     NOT_REFERENCED(Buffer);
62356|     return STATUS_SUCCESS;
62357| }
62358|
62359| /*-----
| -----*/
62360| STATIC NTSTATUS ReadPostProcessSpecialSectors (
62361|     PDEVICE_OBJECT DeviceObject,
62362|     ULARGE_INTEGER LogSector,
62363|     ULONG /*Count*/,
62364|     char *Buffer )
62365| {
62366|     PVDISK_EXTENSION DevExt =
| GetVDiskExtension(DeviceObject);
62367|
62368|     if( (DevExt->IsPhysical) &&
| (LogSector.QuadPart==0)) {
62369|         // this updates the MBR with our special copy
62370|         // (Different Serial Numbers, etc..., see
| devcon.c for more
62371|         // info
62372|         //Debug(DEBUG_READ,("VDisk: Read: Physical
| sector 0 before changes\n"));
62373|         //DumpSector(Buffer,512);

```

```

62374|    ((tMBR*)Buffer)->SerialNumber =
| DevExt->SerialNumber;
62375|    //Debug(DEBUG_READ,("VDisk: Read: Physical
| sector 0 after changes\n"));
62376|    //DumpSector(Buffer,512);
62377|
62378|    //Debug(DEBUG_READ,("VDisk: Read: Modified
| Master Boot Record\n"));
62379|    }
62380|    // if a logical drive then change the serial
| numbers.
62381|
62382|    if(!DevExt->IsPhysical) &&
| (LogSector.QuadPart==0) {
62383|        //Debug(DEBUG_READ,("VDisk: Read:
| -----
| \n"));
62384|        #if DO_ALL_IO
62385|            Debug(DEBUG_READ,("VDisk: Read: Logical
| sector 0 before changes\n"));
62386|            DumpBootSector( Buffer );
62387|            #endif /*DO_ALL_IO*/
62388|
62389|            switch(DevExt->Pi.PartitionType & 0x3f) {
62390|
62391|                // DOS Fat boot sector.
62392|                case PARTITION_XINT13 : // Win95
| partition using extended int13 services
62393|                case PARTITION_FAT_12 :
62394|                case PARTITION_FAT_16 :
62395|                case PARTITION_HUGE :
62396| DoAsFAT16:
62397|                // make sure it is fat... as nt has a
| nasty habit of return type 6
62398|                if
| (strcmp(((PNTFS_BOOT_SECTOR)Buffer)->OemId,"NTFS",4)==0
| ) {
62399|                    goto DoAsNTFS;
62400|                }
62401|                if
| (strcmp(((PFAT32_BOOT_SECTOR)Buffer)->OemId,"FAT32",5)=
| =0) {
62402|                    goto DoAsFAT32;
62403|                }
62404|                // this is a ULONG
62405|                ((PFAT_BOOT_SECTOR)Buffer)->VolumeID =
| (DevExt->SerialNumber & 0xfffff00) | DevExt->Instance;
62406|                DevExt->Cluster0Offset =
| (((PFAT_BOOT_SECTOR)Buffer)->ReservedSectors+
62407|

```

```

| (((PFAT_BOOT_SECTOR)Buffer)->SectorsPerFat*
62408|
| ((PFAT_BOOT_SECTOR)Buffer)->NumberOfFats))+
62409|
| (((PFAT_BOOT_SECTOR)Buffer)->RootDirectory*sizeof(FAT_D
| IR_ENTRY))/DevExt->BPS));
62410|         Debug(DEBUG_READ,("VDisk: Read: Changed
| serial number on FAT12/16 volume '%S' to
| %08x\n",DevExt->Name,((PFAT_BOOT_SECTOR)Buffer)->VolumeI
| D));
62411|         break;
62412|
62413|         // NTFS boot sector
62414|         case PARTITION_IFS      :
62415| DoAsNTFS:
62416|         // make sure it is fat... as nt has a
| nasty habit of return type 6
62417|         if
| (strcmp(((PFAT32_BOOT_SECTOR)Buffer)->OemId,"FAT32",5)=
| =0) {
62418|             goto DoAsFAT32;
62419|         }
62420|         if
| (strcmp(((PFAT_BOOT_SECTOR)Buffer)->OemId,"FAT16",5)==0
| ) {
62421|             goto DoAsFAT16;
62422|         }
62423|         if
| (strcmp(((PFAT_BOOT_SECTOR)Buffer)->OemId,"FAT12",5)==0
| ) {
62424|             goto DoAsFAT16;
62425|         }
62426|         // this is a ULARGE_INTEGER
62427|
| ((PNTFS_BOOT_SECTOR)Buffer)->SerialNumber.LowPart =
| DevExt->Instance;
62428|
| ((PNTFS_BOOT_SECTOR)Buffer)->SerialNumber.HighPart =
| DevExt->SerialNumber;
62429|         DevExt->Cluster0Offset = 0;
62430|         Debug(DEBUG_READ,("VDisk: Read: Changed
| serial number on NTFS volume '%S' to
| %08x%08x\n",DevExt->Name,((PNTFS_BOOT_SECTOR)Buffer)->Se
| rialNumber.HighPart,((PNTFS_BOOT_SECTOR)Buffer)->SerialN
| umber.LowPart));
62431|         break;
62432|
62433|         case PARTITION_FAT32      :
62434|         case PARTITION_FAT32_XINT13 :
62435| DoAsFAT32:

```

```

62436|          // make sure it is fat... as nt has a
| nasty habit of return type 6
62437|          if
| (strcmp(((PNTFS_BOOT_SECTOR)Buffer)->OemId,"NTFS",4)==0
| ) {
62438|          goto DoAsNTFS;
62439|          }
62440|          if
| (strcmp(((PFAT_BOOT_SECTOR)Buffer)->OemId,"FAT16",5)==0
| ) {
62441|          goto DoAsFAT16;
62442|          }
62443|          if
| (strcmp(((PFAT_BOOT_SECTOR)Buffer)->OemId,"FAT12",5)==0
| ) {
62444|          goto DoAsFAT16;
62445|          }
62446|
62447|          // this is a ULONG
62448|          ((PFAT32_BOOT_SECTOR)Buffer)->VolumeID
| = (DevExt->SerialNumber & 0xfffff00) |
| DevExt->Instance;
62449|          DevExt->Cluster0Offset =
| (((PFAT32_BOOT_SECTOR)Buffer)->ReservedSectors+
62450|
| (((PFAT32_BOOT_SECTOR)Buffer)->LargeSectorsPerFat*
62451|
| ((PFAT32_BOOT_SECTOR)Buffer)->NumberOfFats));
62452|          Debug(DEBUG_READ,("VDisk: Read: Changed
| serial number on FAT32 volume '%S' to
| %08x\n",DevExt->Name,((PFAT32_BOOT_SECTOR)Buffer)->Volum
| eID));
62453|          break;
62454|
62455|          case PARTITION_LDM      :
62456|          case PARTITION_EXTENDED :
62457|          case PARTITION_XINT13_EXTENDED : // Same as
| type 5 but uses extended int13 services
62458|          case PARTITION_ENTRY_UNUSED :
62459|          case PARTITION_PREP      :
62460|          case PARTITION_UNIX      :
62461|          case PARTITION_XENIX_1    :
62462|          case PARTITION_XENIX_2    :
62463|          Debug(DEBUG_READ,("VDisk: Read:
| Unsupported partition type %d, Unable to change serial
| numbers on volume
| '%S'\n",DevExt->Pi.PartitionType,DevExt->Name));
62464|          DevExt->Cluster0Offset = 0;
62465|          break;
62466|          default:

```

```

62467|         Debug(DEBUG_READ,("VDisk: Read: Unknown
| Partition type %d, Unable to fudge serial numbers on
| volume '%S'\n",DevExt->Pi.PartitionType,DevExt->Name));
62468|         DevExt->Cluster0Offset = 0;
62469|         break;
62470|     }
62471|
62472|     //Debug(DEBUG_READ,("VDisk: Read: Logical
| sector 0 after changes\n"));
62473|     //DumpSector(Buffer,512);
62474|     //DumpBootSector( Buffer );
62475|     //Debug(DEBUG_READ,("VDisk: Read:
| -----
| \n"));
62476|
62477| }
62478| #if 0
62479| {
62480|     ULONG i;
62481|
62482|     for(i=0;i<Count;i++) {
62483|         Debug(DEBUG_READ,("Logical Sector %l64u,
| Physical Sector
| %d\n",i+LogSector.QuadPart,i+PhySector));
62484|         DumpSector(&Buffer[i*512],512);
62485|     }
62486| }
62487| #endif
62488| return STATUS_SUCCESS;
62489| }
62490|
62491| /*-----*/
| -----*/
62492| STATIC NTSTATUS TdWaitForReadWork()
62493| {
62494|     PVOID ObjectTable[2] = { &VDiskExitingEvent,
| &ReadVDiskSemaphore };
62495|
62496|     ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
62497|     return pmWaitForMultipleObjects(ObjectTable,2,NULL);
62498| }
62499|
62500| /*-----*/
| -----*/
62501| void ReadVDiskThread ( PVOID Context )
62502| {
62503|     ULONG Exiting=0;
62504|     NTSTATUS Status=0;
62505|     ULONG TempBufferSize=GRANULE_SIZE*2;
62506|     PCHAR TempBuffer=NULL;

```

```

62507|
62508| NOT_REFERENCED(Context);
62509| PAGED_CODE();
62510|
62511| pmAcquireMutex ( &VDiskThreadMutex, NULL);
62512| VDiskNumberOfThreads++;
62513| pmReleaseMutex ( &VDiskThreadMutex);
62514|
62515|
    | TempBuffer=(char*)MemAllocatePoolWithTag(PagedPool,TempB
    | ufferSize,PSM_VDISK_BUFFER_TAG);
62516|
62517| RestartThreadFromError:
62518|
62519| __try {
62520|     while(!Exiting) {
62521|         Status = TdWaitForReadWork();
62522|
62523|         if(Status == STATUS_WAIT_1) {
62524|             PLIST_ENTRY ListEntry=NULL;
62525|
62526|             // we should not be at anything other,
        | if APC_LEVEL.
62527|             ASSERT(KeGetCurrentIrql() ==
        | PASSIVE_LEVEL);
62528|
62529| GetAnother:
62530|             ListEntry = ExInterlockedRemoveHeadList
        | (
62531|                 &ReadVDiskQueue,    // List Head
62532|                 &ReadVDiskSpinLock  // Lock
62533|             );
62534|
62535|             if((ListEntry) &&
        | (ListEntry!=&ReadVDiskQueue)) {
62536|                 tReadRequest
        | *ReadRequest=NULL;
62537|                 PVDISK_EXTENSION DevExt=NULL;
62538|                 PIO_STACK_LOCATION
        | currentIrpStack=NULL;
62539|                 CHAR IoIncrement =
        | IO_NO_INCREMENT;
62540|                 char *Buffer=NULL;
62541|                 KIRQL oldIrql;
62542|
62543|                 /*lint -save -e413 */
62544|                 ReadRequest = CONTAINING_RECORD(
        | ListEntry, tReadRequest, ListEntry );
62545|                 /*lint -restore */
62546|

```



```

62547|           DevExt = GetVDiskExtension
        | (ReadRequest->DeviceObject);
62548|
62549|           if(IsBeingProcessedEx(ReadRequest))
        | {
62550|               BOOLEAN OnlyOne=FALSE;
62551|
62552|               pmAcquireSpinLock (
        | &ReadVDiskSpinLock, &oldIrql );
62553|               InsertTailList
        | (&ReadVDiskQueue,&ReadRequest->ListEntry);
62554|
62555|               // if only one on list
62556|               if(ReadVDiskQueue.Flink ==
        | &ReadRequest->ListEntry) {
62557|                   OnlyOne=TRUE;
62558|               }
62559|               pmReleaseSpinLock(
        | &ReadVDiskSpinLock, oldIrql );
62560|               if(OnlyOne) {
62561|                   LARGE_INTEGER TimeToWait =
        | {0};
62562|                   TimeToWait.QuadPart =
        | RELATIVE(MILLISECONDS(1));
62563|                   KeDelayExecutionThread(
        | (KPROCESSOR_MODE)KernelMode, FALSE, &TimeToWait );
62564|               }
62565|               goto GetAnother;
62566|           }
62567|
62568| //           Debug(DEBUG_INFO,("VDisk: Read:
        | Got work, Irp=%08x, De=%08x,
        | '%S'\n",Irp,DeviceObject,DevExt->Name));
62569|
62570|           // make sure PSM is enabled for
        | this device.. otherwise someone is accessing us
62571|           // without our approval...
62572| //
        | ASSERT(((PFILTERED_EXTENSION)(GetDeviceExtension(DevExt-
        | >PSMDevice)))->PSMed);
62573|
62574|           currentIrpStack =
        | IoGetCurrentIrpStackLocation(ReadRequest->Irp);
62575|
62576|           | ReadRequest->Irp->IoStatus.Information = 0;
62577|
62578|           // wait on outstanding requests to
        | finish if user said to.
62579|           // BEFORE we grab any resources!!!

```

```

62580|         if (PSManPSMFlags &
| PSM_FLAG_PAUSE_ON_IO) {
62581|             LARGE_INTEGER TimeToWait;
62582|
62583|             TimeToWait.QuadPart =
| RELATIVE(MICROSECONDS(10));
62584|
62585|             // wait for work to get done...
62586|             while(OutstandingRequests) {
62587|                 KeDelayExecutionThread(
62588|                     (KPROCESSOR_MODE)KernelMode, // IN KPROCESSOR_MODE
| WaitMode,
62589|                     FALSE,    // IN
| BOOLEAN Alertable,
62590|                     &TimeToWait // IN
| PLARGE_INTEGER Interval
62591|                 );
62592|             }
62593|
62594|         }
62595|
62596|
62597|         // acquire snapshot first so we do
| not deadlock
62598|         GetSnapShotForRead();
62599|         // since we are not scanning and
| already have a pointer to the snapshot, we need to
62600|         // make sure that it has not been
| deleted
62601|
62602|         __try {
62603|             if(DevExt->SnapShot) {
62604|
| UseSnapShot(DevExt->SnapShot);
62605|
62606|                 //Debug(DEBUG_READ,("VDisk:
| Read: Acquiring vdisk resource\n"));
62607|                 AcquireVDiskResource();
62608|                 __try {
62609|                     // protect this area so
| we dont bring down NT
62610|                     __try {
62611|
62612|                         // make sure the
| device didnt disappear while
62613|                         // waiting for
| mutex
62614|
| if(DevExt->PartitionActive) {

```

```

62615|
62616|                // if a read
        | (ie not a verify..)
62617|
        | if(currentIrpStack->MajorFunction == IRP_MJ_READ) {
62618| #if DO_ALL_IO
62619|
        | PUNICODE_STRING FileName;
62620|
        | //Debug(DEBUG_READ,("VDisk: Read : Irp %p F=%08x-FO %p
        | F=%08x-Sf=%08x
        | \n",ReadRequest->Irp,ReadRequest->Irp->Flags,ReadRequest
        | ->Irp->Tail.Overlay.OriginalFileObject,currentIrpStack->
        | Flags));
62621|
        | FileName=File_GetFullFileName(ReadRequest->Irp->Tail.Ove
        | rlay.OriginalFileObject);
62622|
        | Debug(DEBUG_READ,("VDisk: Read : %p-%p Log (%p)=%l64x
        | for %03x
        | '%wZ'\n",ReadRequest->Irp,ReadRequest->Irp->Tail.Overlay
        | .OriginalFileObject,ReadRequest->DeviceObject,ReadReques
        | t->RealSector,ReadRequest->RealCount,FileName));
62623|
        | if(FileName) {
62624|
        | FREE_POINTER(FileName);
62625|                }
62626| #endif
62627|                if (
        | ReadRequest->Irp->MdlAddress != NULL ) {
62628|                ULONG
        | Info;
62629|
62630|
62631|                // we
        | are not going to use our caching logic on the reads
        | because
62632|                // 1.
        | The hard drive is plenty fast
62633|                // 2.
        | NT is caching the hard drive, so we would only
        | duplicate, and not
62634|                //
        | get very many hits.
62635|
62636|                //
        | empty function so dont waste cycles for now.
62637|
        | //ReadPreProcessSpecialSectors(

```

```

    | ReadRequest->DeviceObject, LogSector, Count, Buffer );
62638|
62639|                // this
    | routine will read from the device with out double
    | mapping
62640|                // the
    | address
62641|                //
    | 2-5-99 Make sure this stays as is!
62642|                // This
    | solves the problem where when "reading" from a file, NT
    | cache manager
62643|                //
    | would "write" back to it. The reason for this is a new
    | "Mdl" would describe
62644|                // the
    | range. When the Mdl is freed, it is marked dirty so
    | the original "Mdl"
62645|                // gets
    | written back to.
62646|                Status
    | = Sblo_ReadDeviceMdl(
62647|    | GetFilteredExtension(DevExt->PSMDevice)->TargetDeviceObj
    | ect,
62648|    | &ReadRequest->ByteOffset,
62649|    | ReadRequest->ByteLength,
62650|    | ReadRequest->Irp, ReadRequest->Irp->MdlAddress);
62651|
62652|                // but
    | as long as we got the revert, we need to double map.
62653| #if _WIN32_WINNT >=0x0500
62654|                Buffer
    | = (char *)MmGetSystemAddressForMdlSafe(
    | ReadRequest->Irp->MdlAddress, NormalPagePriority );
62655| #else
62656|                Buffer
    | = (char *)MmGetSystemAddressForMdl(
    | ReadRequest->Irp->MdlAddress );
62657| #endif
62658|    | ASSERT(Buffer);
62659|
62660|
    | if(!Buffer) {
62661|
    | Status = STATUS_INSUFFICIENT_RESOURCES;

```

```

62662|                                     }
62663|
62664|
62665|     | if(NT_SUCCESS(Status)) {
62666|     | PCHAR BufferToUse=Buffer;
62667|                                     /*
62668|
62669|     | 1-12-2001 - rob -
62670|
62671|     | Since we did the Sblo_ReadDeviceMdl above, and we also
62672|     | got a system virtual address for it,
62673|
62674|     | When we send it to SblInternalRevertBuffer and the data
62675|     | is in the cache file, The Virtual address
62676|
62677|     | will be sent to the filesystem which then gets another
62678|     | Mdl for the virtual address. We now have 2 Mdls
62679|
62680|     | pointing to the same physical memory. When the second
62681|     | one is freed (This is speculation here) the physical
62682|
62683|     | pages are marked dirty, but since there is another Mdl,
62684|     | the reference count for the pages is not zero, not
62685|     | causing
62686|
62687|     | the pages to be freed. So some time later when the
62688|     | MiMappedPageWriter thread runs he writes the "dirty"
62689|     | pages
62690|
62691|     | back to the files, which is superfluous. The crux of
62692|     | this is that when this happens, a virtual write occurs
62693|     | wasting space in
62694|
62695|     | the cache file.
62696|
62697|
62698|     | The workaround is to use another buffer to have the
62699|     | file system fill in stuff from our cache file and then
62700|     | merge the
62701|
62702|     | two together. If not enough memory for the cache file
62703|     | read, then we will use the actual buffer (which will
62704|     | then generate
62705|
62706|     | the extra virtual write). This case shouldnt happen
62707|     | often.
62708|                                     */
62709| #define SectorSize (DevExt->BPS)

```

```

62683|
| if(ReadRequest->RealCount*DevExt->BPS>TempBufferSize) {
62684|
| TempBufferSize =
| ROUND_UP(ReadRequest->RealCount,SECTORS_PER_GRANULE)*Dev
| Ext->BPS;
62685| #undef SectorSize
62686|
| if(TempBuffer) {
62687|
| FREE_POINTER(TempBuffer);
62688|
| }
62689| TryAllocTempBuffer:
62690|
| TempBuffer =
| (char*)MemAllocatePoolWithTag(PagedPool,TempBufferSize,P
| SM_VDISK_BUFFER_TAG);
62691|
| if(TempBuffer) {
62692|
| BufferToUse = TempBuffer;
62693|
| }
62694|                                     }
| else
62695|
| if(TempBuffer) {
62696|
| BufferToUse=TempBuffer;
62697|                                     }
| else {
62698|
| goto TryAllocTempBuffer;
62699|                                     }
62700|
62701|                                     //
| if any changes to the filtered drive, put back the
| original unchanged
62702|                                     //
| data
62703|
| if(BufferToUse==TempBuffer) {
62704|
| RtlMoveMemory(BufferToUse,Buffer,ReadRequest->RealCount*
| DevExt->BPS);
62705|                                     }
62706|
| Status = SbInternalRevertBuffer(
62707|

```

```

    | DevExt->PSMDevice,
62708|
    | DevExt,
62709|
    | ReadRequest->RealSector,
62710|
    | ReadRequest->RealCount,
62711|
    | FALSE,
62712|
    | BufferToUse,
62713|
    | &Info);
62714|                                     //
    | if Info==0 then no changes occurred, otherwise number of
    | sectors changed
62715|
    | if((Info!=0) && (BufferToUse==TempBuffer)) {
62716|
    | RtlMoveMemory(Buffer,BufferToUse,ReadRequest->RealCount*
    | DevExt->BPS);
62717|                                     }
62718|                                     } else
    | {
62719|
    | Debug(DEBUG_READ,("VDisk: Read: Error %08x reading from
    | device\n",Status));
62720|                                     }
62721|
    | //Status = Cache_ReadSector( &VDiskCache,
    | DevExt->PhysicalDevice, LogSector, Count, Buffer );
62722|
    | if(NT_SUCCESS(Status)) {
62723|
    | IoIncrement = IO_DISK_INCREMENT;
62724|
    | ReadRequest->Irp->IoStatus.Information =
    | ReadRequest->ByteLength;
62725|
    | ReadPostProcessSpecialSectors (
62726|
    | ReadRequest->DeviceObject,
62727|
    | ReadRequest->RealSector,
62728|
    | ReadRequest->RealCount,
62729|
    | Buffer );
62730|
62731|

```

```

        | //Debug(DEBUG_READ,("VDisk: Read: Log (%p)=%d for
        | %d\n",DeviceObject,LogSector,Count));
62732|
        | //DumpSector(Buffer,currentIrpStack->Parameters.Read.Len
        | gth);
62733|
62734|                                     //
        | Update stats about volume
62735|                                     //
        | dont need spin lock as only 1 read can occur at any
        | time
62736|
        | DevExt->SectorsRead+=ReadRequest->RealCount;
62737|
        | DevExt->NumberOfReadRequests++;
62738|                                     } else
        | {
62739|
        | Debug(DEBUG_READ,("VDisk: Read: Error %08x reading from
        | device (or cache file)\n",Status));
62740|                                     }
62741|                                     } else {
62742|
        | Debug(DEBUG_READ,("VDisk: Read: Invalid MDL\n"));
62743|                                     Status
        | = STATUS_INVALID_USER_BUFFER;
62744|                                     }
62745|                                     } else {
62746|
        | Debug(DEBUG_READ,("VDisk: Verify: Log (%p)=%l64d for
        | %d\n",
62747|
        | ReadRequest->DeviceObject,
62748|
        | ReadRequest->RealSector.QuadPart,
62749|
        | ReadRequest->RealCount));
62750|                                     Status =
        | STATUS_SUCCESS;
62751|
        | ReadRequest->Irp->IoStatus.Information =
        | ReadRequest->ByteLength;
62752|                                     }
62753|                                     } else {
62754|
        | Debug(DEBUG_READ,("VDisk: Read: Device
        | disappeared\n"));
62755|                                     Status =
        | STATUS_NO_MEDIA_IN_DEVICE;
62756|

```



```

    | ReadRequest->Irp->IoStatus.Information = 0;
62757|                }
62758|
62759|                // if media error,
    | say so...
62760|                switch (Status) {
62761|                case
    | STATUS_MEDIA_CHANGED :
62762|                case
    | STATUS_NO_MEDIA_IN_DEVICE : {
62763|
    | InterlockedIncrement((PLONG)&DevExt->DiskChangeCount);
62764|
    | DevExt->DriveNotReady = TRUE;
62765|
62766|
    | Debug(DEBUG_READ,("VDisk: Read: Media may have
    | changed\n"));
62767|                if (
    | DevExt->DeviceObject->Vpb->Flags & VPB_MOUNTED ) {
62768|
    | Debug(DEBUG_READ,("VDisk: Read: Informing File system
    | Me=%08x, DO=%08x,
    | RO=%08x\n",DevExt->DeviceObject,DevExt->DeviceObject->Vp
    | b->DeviceObject,DevExt->DeviceObject->Vpb->RealDevice));
62769|
    | DevExt->DeviceObject->Flags |= DO_VERIFY_VOLUME;
62770|                Status
    | = STATUS_VERIFY_REQUIRED;
62771|                } else {
62772|                // we
    | may need to do this.
62773|                //
    | Status = STATUS_IO_DEVICE_ERROR;
62774|                }
62775|                break;
62776|                }
62777|                default:
62778|                break;
62779|                }
62780|                }
    | __except(ExceptionFilter(GetExceptionInformation())) {
62781|                Status =
    | GetExceptionCode();
62782|
    | Debug(DEBUG_READ,("VDisk: Read: Exception
    | %08x\n",Status));
62783|                }
62784|                } __finally {
62785|                ReleaseVDiskResource();

```

```

62786|
| ReadRequest->Irp->IoStatus.Status = Status;
62787| // TESTTEST Keep "Media ejected, please
| reinsert" from occurring
62788| #if 1
62789|
| if(!IoIsErrorUserInduced(Status)) {
62790|
| Debug(DEBUG_READ,("Setting hard error for error
| %08x\n",Status));
62791|
| IoSetHardErrorOrVerifyDevice(ReadRequest->Irp,DevExt->De
| viceObject);
62792| }
62793| #endif
62794| if(Status!=0) {
62795|
| Debug(DEBUG_READ,("VDisk: Read: Device %08x Irp %08x
| Error
| %08x\n",DevExt->DeviceObject,ReadRequest->Irp,Status));
62796|
| }
62797| }
62798| IoCompleteRequest
| (ReadRequest->Irp, IoIncrement) ;
62799| }
62800| } else {
62801| Debug(DEBUG_DCPSM,("Read:
| SnapShot has been deleted while waiting or not
| psmcd\n"));
62802| Status =
| STATUS_NO_MEDIA_IN_DEVICE;
62803| }
62804| } __finally {
62805| if(DevExt->SnapShot) {
62806|
| DoneWithSnapShot(DevExt->SnapShot);
62807| }
62808| ReleaseSnapShotForRead();
62809| }
62810|
| pmAcquireSpinLock(&WriteSpinLock,&oldIrql);
62811|
| RemoveEntryList(&(ReadRequest->ProcessingEntry));
62812|
| pmReleaseSpinLock(&WriteSpinLock,oldIrql);
62813| FREE_POINTER(ReadRequest);
62814| } else {
62815| Debug(DEBUG_INFO,("VDisk: Read:
| Error ListEntry is NULL\n"));
62816| }

```

```

62817|         } else {
62818|             Exiting = 1;
62819|             Debug(DEBUG_INFO,("VDisk: Read: Error
| Status = %08x\n",Status));
62820|         }
62821|     }
62822| }
| __except(ExceptionFilter(GetExceptionInformation())) {
62823|     Debug(DEBUG_READ,("VDisk: Read: Exception %08x
| in thread\n",GetExceptionCode()));
62824| }
62825|
62826| // cant goto from within exception handler...
62827| if(!Exiting) {
62828|     goto RestartThreadFromError;
62829| }
62830|
62831| //ExitThread:
62832|
62833|
62834| //ExitThreadFinal:
62835|
62836| if(TempBuffer) {
62837|     FREE_POINTER(TempBuffer);
62838| }
62839| Debug(DEBUG_READ,("VdiskRead: Exiting\n"));
62840| AcquireVDiskResource();
62841| __try {
62842|     // free any writes to the volume
62843|     while(!IsListEmpty(&ReadVDiskQueue)) {
62844|         PLIST_ENTRY ListEntry=NULL;
62845|         tReadRequest *ReadRequest=NULL;
62846|         PIRP Irp=NULL;
62847|         KIRQL oldIrql;
62848|
62849|         Debug(DEBUG_READ,("VdiskRead: Cleaning up
| read on vdisk queue\n"));
62850|         ListEntry = ExInterlockedRemoveHeadList (
62851|             &ReadVDiskQueue,    // List Head
62852|             &ReadVDiskSpinLock  // Lock
62853|         );
62854|         /*lint -save -e413 */
62855|         ReadRequest = CONTAINING_RECORD( ListEntry,
| tReadRequest, ListEntry );
62856|         /*lint -restore */
62857|
62858|         Irp      = ReadRequest->Irp;
62859|         pmAcquireSpinLock(&WriteSpinLock,&oldIrql);
62860|
| RemoveEntryList(&(ReadRequest->ProcessingEntry));

```

```

62861|         pmReleaseSpinLock(&WriteSpinLock,oldIrql);
62862|         FREE_POINTER(ReadRequest);
62863|
62864|         Irp->IoStatus.Information = 0;
62865|         Irp->IoStatus.Status =
        | STATUS_NO_MEDIA_IN_DEVICE;
62866|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
62867|     }
62868| } __finally {
62869|     ReleaseVDiskResource();
62870| }
62871|
62872|
62873| pmAcquireMutex ( &VDiskThreadMutex, NULL);
62874| VDiskNumberOfThreads--;
62875| pmReleaseMutex ( &VDiskThreadMutex);
62876|
62877| Debug(DEBUG_READ,("VdiskRead: Exited\n"));
62878| PsTerminateSystemThread( 0 );
62879| }
62880|
62881| STATIC NTSTATUS FillBufferWithCompressableData( PCHAR
        | Buffer, ULONG size )
62882| {
62883|     CHAR b=0;
62884|
62885|     while(size--) {
62886|         *Buffer++ = b++;
62887|     }
62888|     return 0;
62889| }
62890|
62891| /*-----
        | -----*/
62892| STATIC NTSTATUS PSMANReadVDisk(
62893|     IN PDEVICE_OBJECT DeviceObject,
62894|     IN PIRP Irp
62895| )
62896| {
62897|     NTSTATUS Status=STATUS_PENDING;
62898|     tReadRequest *ReadRequest=NULL;
62899|     PVDISK_EXTENSION DevExt =
        | GetVDiskExtension(DeviceObject);
62900|
62901| //     Debug(DEBUG_READ | DEBUG_PROCCALL,
        | ("PSMANReadVDisk Called\n"));
62902|
62903|     if (CheckMediaLoaded(DeviceObject, Irp
        | )!=STATUS_SUCCESS) {
62904|         Debug(DEBUG_READ,("VDisk: Read: Media not

```

```

    | loaded\n"));
62905|     Status = Irp->IoStatus.Status;
62906|     IoCompleteRequest(Irp, 0);
62907|     //Debug(DEBUG_READ | DEBUG_PROCCALL,
    | ("PSManReadVDisk Done %08x\n",Status));
62908|     return Status;
62909| }
62910|
62911| // if we get here, something is wrong..
62912| if(!VDiskNumberOfThreads) {
62913|     Debug(DEBUG_READ | DEBUG_PROCCALL,
    | ("PSManReadVDisk: No threads to handle request!!!\n"));
62914|     Irp->IoStatus.Information = 0;
62915|     Status = Irp->IoStatus.Status =
    | STATUS_INVALID_DEVICE_STATE;
62916|     IoCompleteRequest(Irp, 0);
62917|     return Status;
62918| }
62919|
62920| const ULONG FillCode = gVDiskIOHandling &
    | PSM_VDISK_FLAG_FILL_MASK;
62921| if(FillCode) {
62922|     Irp->IoStatus.Information =
    | IoGetCurrentIrpStackLocation(Irp)->Parameters.Read.Length
    | h;
62923|     if(FillCode == PSM_VDISK_FLAG_BUFFER_NO_FILL) {
62924|         // leave buffer alone
62925|     } else
62926|     if(FillCode ==
    | PSM_VDISK_FLAG_BUFFER_FILL_WITH_ZEROES) {
62927|         // fill with zeros
62928|         RtlZeroMemory(MmGetSystemAddressForMdl(
    | Irp->MdlAddress ),Irp->IoStatus.Information);
62929|     } else
62930|     if(FillCode ==
    | PSM_VDISK_FLAG_BUFFER_FILL_COMPRESS) {
62931|         // fill with compressable data
62932|         FillBufferWithCompressableData((char
    | *)MmGetSystemAddressForMdl( Irp->MdlAddress
    | ),Irp->IoStatus.Information);
62933|     }
62934|     Status = Irp->IoStatus.Status = STATUS_SUCCESS;
62935|     IoCompleteRequest(Irp, 0);
62936|     return Status;
62937| }
62938|
62939| ReadRequest = (tReadRequest
    | *)MemAllocatePoolWithTag(NonPagedPool,
    | sizeof(tReadRequest),READREQUESTTAG);
62940| if(ReadRequest) {

```

```

62941|         IoMarkIrpPending(Irp);
62942|
62943|         FillInWriteRequest( ReadRequest, DeviceObject,
        | Irp, DevExt->BPS);
62944|
62945|         // add to queue...
62946|         ExInterlockedInsertTailList ( &ReadVDiskQueue,
62947|         | &(ReadRequest->ListEntry),
62948|         | &ReadVDiskSpinLock);
62949|
62950|
62951|         pmSignalSemaphore( &ReadVDiskSemaphore);
62952|
62953|     } else {
62954|         Debug(DEBUG_READ,("VDisk: Read: Out of memory
        | for read command\n"));
62955|         | Status=Irp->IoStatus.Status=STATUS_INSUFFICIENT_RESOURCE
        | S;
62956|         Irp->IoStatus.Information = 0;
62957|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
62958|     }
62959|
62960|     //Debug(DEBUG_READ | DEBUG_PROCCALL,
        | ("PSManReadVDisk Done %08x\n",Status));
62961|     return Status;
62962| }
62963|
62964|
62965| /*-----
        | -----*/
62966| STATIC NTSTATUS
62967| PSManReadFSObject(
62968|     IN PDEVICE_OBJECT DeviceObject,
62969|     IN PIRP Irp
62970| )
62971|
62972| /*++
62973|
62974| Routine Description:
62975|
62976| This is the driver entry point for read requests
62977| to disks to which the PSMan driver has attached.
62978| This driver collects statistics and then sets a
        | completion
62979| routine so that it can collect additional
        | information when
62980| the request completes. Then it calls the next

```

```

| driver below
62981|   it.
62982|
62983| Arguments:
62984|
62985|   DeviceObject
62986|   Irp
62987|
62988| Return Value:
62989|
62990|   NTSTATUS
62991|
62992| --*/
62993|
62994| {
62995|   NTSTATUS Status=STATUS_INVALID_PARAMETER;
62996|   NOT_REFERENCED(DeviceObject);
62997|
62998|   Debug(DEBUG_PROCCALL,("PSManReadFSObject
| Called\n"));
62999|   Irp->IoStatus.Status = Status;
63000|   Irp->IoStatus.Information = 0;
63001|
63002|   IoCompleteRequest(Irp, IO_NO_INCREMENT);
63003|   Debug(DEBUG_PROCCALL,("PSManReadFSObject Done\n"));
63004|   return Status;
63005| } // PSManReadFSObject
63006|
63007|
63008| /*-----
| -----*/
63009| STATIC NTSTATUS
63010| PSManReadFSFilter(
63011|   IN PDEVICE_OBJECT DeviceObject,
63012|   IN PIRP Irp
63013|   )
63014|
63015| /*++
63016|
63017| Routine Description:
63018|
63019|   Pass irp to handler
63020|
63021| Arguments:
63022|
63023|   DriverObject - Pointer to device object to being
| shutdown by system.
63024|   Irp          - IRP involved.
63025|
63026| Return Value:

```

```

63027|
63028|    NT Status
63029|
63030| --*/
63031|
63032| {
63033|    NTSTATUS Status;
63034| /*
63035| #ifdef DEBUG
63036|    if(PsmActive) {
63037|        Debug(DEBUG_READ |
        | DEBUG_PROCCALL,("PSManReadFSFilter Called Device=%p,
        | Irp=%p\n",DeviceObject,Irp));
63038|    }
63039| #endif
63040| */
63041|    Status = PSManFSPassThru( DeviceObject, Irp );
63042| /*
63043| #ifdef DEBUG
63044|    if(PsmActive) {
63045|        Debug(DEBUG_READ |
        | DEBUG_PROCCALL,("PSManReadFSFilter Done   Device=%p,
        | Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
63046|    }
63047| #endif
63048| */
63049|    return Status;
63050| } // end PSManReadFSFilter()
63051|
63052|
63053|
63054| File Listing: READ.h
63055|
63056| NTSTATUS
63057| PSManRead(
63058|    IN PDEVICE_OBJECT DeviceObject,
63059|    IN PIRP Irp
63060| );
63061|
63062| STATIC NTSTATUS
63063| PSManReadObject(
63064|    IN PDEVICE_OBJECT DeviceObject,
63065|    IN PIRP Irp
63066| );
63067|
63068| STATIC NTSTATUS
63069| PSManReadDevice(
63070|    IN PDEVICE_OBJECT DeviceObject,
63071|    IN PIRP Irp
63072| );

```



```

63073|
63074| STATIC NTSTATUS
63075| PSMANReadVDisk(
63076|     IN PDEVICE_OBJECT DeviceObject,
63077|     IN PIRP Irp
63078| );
63079|
63080| STATIC NTSTATUS
63081| PSMANReadFSObject(
63082|     IN PDEVICE_OBJECT DeviceObject,
63083|     IN PIRP Irp
63084| );
63085| STATIC NTSTATUS
63086| PSMANReadFSFilter(
63087|     IN PDEVICE_OBJECT DeviceObject,
63088|     IN PIRP Irp
63089| );
63090|
63091| void ReadVDiskThread ( PVOID Context );
63092|
63093|
63094|
63095| File Listing: REG.cpp
63096|
63097| #include "precomp.h"
63098|
63099| /*-----
    | -----*/
63100| void Reg_GetULONGKey (
63101|     IN PUNICODE_STRING RegistryPath,
63102|     IN PWCHAR Key,
63103|     IN ULONG Default,
63104|     OUT PULONG Result
63105| )
63106| /*++
63107|
63108| Routine Description:
63109|
63110|     Get a ULONG from the registry
63111|
63112| Arguments:
63113|
63114|     RegistryPath  the path to query
63115|     Key           Key to query
63116|     Default       Default value to return if Key does
    | not exist
63117| Return Value:
63118|
63119|     The Key value
63120|

```

```

63121| --*/
63122|
63123| {
63124|     RTL_QUERY_REGISTRY_TABLE paramTable[2]={0};
63125|     PWCHAR path=NULL;
63126|
63127|     //
63128|     // Since the registry path parameter is a "counted"
        | UNICODE string, it
63129|     // might not be zero terminated. For a very short
        | time allocate memory
63130|     // to hold the registry path zero terminated so
        | that we can use it to
63131|     // delve into the registry.
63132|     //
63133|
63134|     path = (PWCHAR) MemAllocateString(256);
63135|
63136|     if (path) {
63137|
63138|         RtlZeroMemory( &paramTable[0],
            | sizeof(paramTable) );
63139|         RtlZeroMemory( path,
            | RegistryPath->Length+sizeof(WCHAR) );
63140|         RtlMoveMemory( path, RegistryPath->Buffer,
            | RegistryPath->Length );
63141|
63142|         paramTable[0].Flags =
            | RTL_QUERY_REGISTRY_DIRECT;
63143|         paramTable[0].Name = Key;
63144|         paramTable[0].EntryContext = Result;
63145|         paramTable[0].DefaultType = REG_DWORD;
63146|         paramTable[0].DefaultData = &Default;
63147|         paramTable[0].DefaultLength = sizeof(ULONG);
63148|
63149|         if (!NT_SUCCESS(RtlQueryRegistryValues(
63150|             RTL_REGISTRY_ABSOLUTE |
            | RTL_REGISTRY_OPTIONAL,
63151|             path,
63152|             &paramTable[0],
63153|             NULL,
63154|             NULL
63155|             ))) {
63156|             *Result = Default;
63157|         }
63158|
63159|         // We don't need that path anymore.
63160|         MemFreeString(path);
63161|     } else {
63162|         Debug(DEBUG_INFO,("Psman: Reg_GetULONGKey: Out

```

```

    | of memory\n"));
63163| }
63164| }
63165|
63166| /*-----
    | -----*/
63167| void Reg_GetStringKey (
63168|     IN PUNICODE_STRING RegistryPath,
63169|     IN PWCHAR Key,
63170|     IN PWCHAR Default,
63171|     OUT PUNICODE_STRING Result
63172| )
63173| /*++
63174|
63175| Routine Description:
63176|
63177|     Get a string from the registry
63178|
63179| Arguments:
63180|
63181|     RegistryPath  the path to query
63182|     Key           Key to query
63183|     Default       Default value to return if Key does
        | not exist
63184| Return Value:
63185|
63186|     The Key value
63187|
63188| --*/
63189|
63190| {
63191|     RTL_QUERY_REGISTRY_TABLE paramTable[2]={0};
63192|     PWCHAR path=NULL;
63193|     NTSTATUS Status;
63194|
63195|     //
63196|     // Since the registry path parameter is a "counted"
        | UNICODE string, it
63197|     // might not be zero terminated. For a very short
        | time allocate memory
63198|     // to hold the registry path zero terminated so
        | that we can use it to
63199|     // delve into the registry.
63200|     //
63201|
63202|     path = (PWCHAR) MemAllocatePoolWithTag( PagedPool,
        | RegistryPath->Length+sizeof(WCHAR),REGISTRYTAG);
63203|
63204|     if (path) {
63205|

```

```

63206|     RtlZeroMemory( &paramTable[0],
| sizeof(paramTable) );
63207|     RtlZeroMemory( path,
| RegistryPath->Length+sizeof(WCHAR) );
63208|     RtlMoveMemory( path, RegistryPath->Buffer,
| RegistryPath->Length );
63209|
63210|     paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT
| | RTL_QUERY_REGISTRY_NOEXPAND;
63211|     paramTable[0].Name = Key;
63212|     paramTable[0].EntryContext = Result;
63213|     paramTable[0].DefaultType = REG_SZ;
63214|     paramTable[0].DefaultData = Default;
63215|     paramTable[0].DefaultLength =
| wcslen(Default)*sizeof(WCHAR)+sizeof(WCHAR);
63216|
63217|     Result->Length = 0;
63218|     Result->MaximumLength=256;
63219|     Result->Buffer = (WCHAR
| *)MemAllocatePoolWithTag(PagedPool,256,REGISTRYTAG);
63220|
63221|     Debug(DEBUG_INFO,("Path='%S',
| Key='%S'\n",path,Key));
63222|
63223|     if (!NT_SUCCESS(Status =
| RtlQueryRegistryValues(
63224|         RTL_REGISTRY_ABSOLUTE |
| RTL_REGISTRY_OPTIONAL,
63225|         path,
63226|         &paramTable[0],
63227|         NULL,
63228|         NULL
63229|     ))) {
63230|
63231|         Debug(DEBUG_INFO,("Error %08x reading
| registry key '%S'\n",Status, path));
63232|         if(Result->Buffer) {
63233|
| RtlZeroMemory(Result->Buffer,Result->MaximumLength);
63234|         Result->Length =
| wcslen(Default)*sizeof(WCHAR);
63235|
| RtlCopyMemory(Result->Buffer,Default,Result->Length);
63236|         Debug(DEBUG_INFO,("Setting default of
| '%S'\n",Result->Buffer));
63237|     } else {
63238|         Debug(DEBUG_INFO,("Error! Out of
| memory\n"));
63239|     }
63240| } else {

```

```

63241|
    | Debug(DEBUG_INFO,("Value=%S\n",Result->Buffer));
63242|     }
63243|
63244|     // We don't need that path anymore.
63245|     FREE_POINTER(path);
63246| } else {
63247|     Debug(DEBUG_INFO,("Psman: Reg: GetUlong: Out of
    | memory\n"));
63248| }
63249| }
63250|
63251| void Reg_FreeString( PUNICODE_STRING String )
63252| {
63253|     String->Length = String->MaximumLength = 0;
63254|     FREE_POINTER(String->Buffer);
63255|     return;
63256| }
63257|
63258| NTSTATUS QueryValue( PVOID KeyHandle,
63259|                     WCHAR *KeyName,
63260|                     PKEY_VALUE_FULL_INFORMATION
    | Value,
63261|                     ULONG *ValueSize)
63262| {
63263|     UNICODE_STRING UniName;
63264|     ULONG DataSize;
63265|     ULONG Err;
63266|
63267|     RtlInitUnicodeString(&UniName,KeyName);
63268|     Err = ZwQueryValueKey(
63269|         KeyHandle,
63270|         &UniName,
63271|         KeyValueFullInformation,
63272|         Value,
63273|         *ValueSize,
63274|         &DataSize
63275|     );
63276|
63277|     *ValueSize = DataSize;
63278|     return Err;
63279| }
63280|
63281|
63282| NTSTATUS Reg_GetBinaryKey(
63283|     IN PUNICODE_STRING RegistryPath,
63284|     IN PWCHAR Key,
63285|     OUT PVOID *Buffer,
63286|     OUT PVOID *Handle
63287| )

```

```

63288| {
63289|     NTSTATUS Status;
63290|     PVOID KeyHandle;
63291|     OBJECT_ATTRIBUTES ObjAttr;
63292|
63293|     *Buffer = NULL;
63294|
63295|     | InitializeObjectAttributes(&ObjAttr,RegistryPath,OBJ_CAS
        | E_INSENSITIVE,NULL,NULL);
63296|     Status = ZwOpenKey( &KeyHandle, KEY_ALL_ACCESS,
        | &ObjAttr );
63297|
63298|     if(NT_SUCCESS(Status)) {
63299|         ULONG
        | DataSize=sizeof(KEY_VALUE_FULL_INFORMATION);
63300|         KEY_VALUE_FULL_INFORMATION Size;
63301|         KEY_VALUE_FULL_INFORMATION *Data;
63302|
63303|         Status =
        | QueryValue(KeyHandle,Key,&Size,&DataSize);
63304|         if(Status==STATUS_BUFFER_OVERFLOW) {
63305|             Data = (KEY_VALUE_FULL_INFORMATION
        | *)MemAllocatePoolWithTag(PagedPool,DataSize,REGISTRYTAG)
        | ;
63306|             if(Data) {
63307|                 PCHAR BinData;
63308|                 Status =
        | QueryValue(KeyHandle,Key,Data,&DataSize);
63309|                 if(NT_SUCCESS(Status)) {
63310|                     | BinData=((PCHAR)Data)+Data->DataOffset;
63311|                     Debug(DEBUG_REG,("Reg_GetBinaryKey:
        | Data offset=%08x, size = %08x,
        | buffer=%08x\n",Data->DataOffset,Data->DataLength,BinData
        | ));
63312|                     *Buffer=BinData;
63313|                     *Handle=Data;
63314|                 } else {
63315|                     Debug(DEBUG_REG,("Reg_GetBinaryKey:
        | Error %08x getting binary value from
        | registry\n",Status));
63316|                 }
63317|             } else {
63318|                 Status = STATUS_INSUFFICIENT_RESOURCES;
63319|                 Debug(DEBUG_REG,("Reg_GetBinaryKey: out
        | of memory\n",Status));
63320|             }
63321|         }
63322|         ZwClose(KeyHandle);

```

```

63323| } else {
63324|     Debug(DEBUG_REG,("Reg_GetBinaryKey: Error %08x
| opening key\n",Status));
63325| }
63326| return Status;
63327| }
63328|
63329| void Reg_FreeBinary( PVOID Handle )
63330| {
63331|     MemFreePool(Handle);
63332|     return;
63333| }
63334|
63335|
63336|
63337| File Listing: REG.h
63338|
63339| void Reg_GetULONGKey (
63340|     IN PUNICODE_STRING RegistryPath,
63341|     IN PWCHAR Key,
63342|     IN ULONG Default,
63343|     OUT PULONG Result
63344| );
63345|
63346| void Reg_GetStringKey (
63347|     IN PUNICODE_STRING RegistryPath,
63348|     IN PWCHAR Key,
63349|     IN PWCHAR Default,
63350|     OUT PUNICODE_STRING Result
63351| );
63352| void Reg_FreeString( PUNICODE_STRING String );
63353|
63354| NTSTATUS Reg_GetBinaryKey(
63355|     IN PUNICODE_STRING RegistryPath,
63356|     IN PWCHAR Key,
63357|     OUT PVOID *Buffer,
63358|     OUT PVOID *Handle
63359| );
63360| void Reg_FreeBinary( PVOID Binary);
63361|
63362|
63363|
63364| File Listing: resource.h
63365|
63366| //{NO_DEPENDENCIES}
63367| // Microsoft Developer Studio generated include file.
63368| // Used by SBPSMAN.RC
63369| //
63370| #define VER_PRODUCTBUILD          3
63371| #define VER_PRODUCTVERSION_W      0x0101

```

```

63372|
63373| // Next default values for new objects
63374| //
63375| #ifdef APSTUDIO_INVOKED
63376| #ifndef APSTUDIO_READONLY_SYMBOLS
63377| #define _APS_NO_MFC                1
63378| #define _APS_NEXT_RESOURCE_VALUE    101
63379| #define _APS_NEXT_COMMAND_VALUE     40001
63380| #define _APS_NEXT_CONTROL_VALUE     1000
63381| #define _APS_NEXT_SYMED_VALUE       101
63382| #endif
63383| #endif
63384|
63385|
63386|
63387| File Listing: revert.cpp
63388|
63389| #include "precomp.h"
63390|
63391| #define RevertDebug(x) Debug(DEBUG_DCPSM,x)
63392|
63393| STATIC BOOLEAN Global_InRevert = FALSE;
63394| STATIC BOOLEAN Global_NeedResultUpdate = FALSE;
63395| STATIC ULONG IsSystemVolume ( const WCHAR *DeviceName
    | );
63396|
63397| //-----
    | -----
63398|
63399| struct RevertThreadParms {
63400|     pkSnapshotMaster    Master;          // IN:
        | snapshot master to revert to
63401|     ULONG               Flags;          // IN:
        | revert flags (see revert.h)
63402|     ULARGE_INTEGER       StartingGranule; // IN:
        | granule to start at on starting volume
63403|     PDEVICE_OBJECT       VolumeDeviceObject; // IN:
        | volume to revert
63404|
        | //-----
        | -----
63405|     NTSTATUS             Status;          // OUT:
        | result of revert operation
63406|     ULONG               RevertUndoSequence; // OUT:
        | master created for revert undo
63407| };
63408|
63409| //-----
    | -----
    | -

```



```

63410|
63411| struct VolumeEntry {
63412|     pkSnapShotEntry    snapshot;
63413|     ULONG               volumeld;
63414|     NTSTATUS            setNameStatus;
        | // result of calling SbSetUserName
63415|     ULARGE_INTEGER      cacheSlotsUsed;
        | // used in pre-test of revert to determine cache usage
63416|     ULONG               saveOpenCloseAcquired;
        | // saves state of devExt->OpenCloseAcquired
63417|     PDEVICE_OBJECT      volumeObject;
        | // volume device object for original filtered device
63418|     ULONG               dismantled:1;
        | // is the volume currently dismantled?
63419|     ULONG               locked:1;
        | // is the volume currently locked?
63420|     ULONG               acquired:1;
        | // whether or not we need to do DoneWithSnapShot
63421|     ULONG               openCloseDirty:1;
        | // whether or not we need to do
        | 'devExt->OpenCloseAcquired = saveOpenCloseAcquired;'
63422|     ULONG               dismantledFailed:1;
        | // only true if we tried to dismantl volume but it
        | failed.
63423|     ULONG               isSystemVolume:1;
        | // true if the volume is what we boot O/S from.
63424| };
63425|
63426| //-----
        | -----
        | -
63427|
63428| NTSTATUS UpdateRevertStatus    ( DWORD StatusCode );
63429| NTSTATUS UpdateLastRevertResult ( DWORD ErrorCode );
63430|
63431| NTSTATUS UpdateRevertRecoveryInfo (
63432|     PDEVICE_OBJECT    Volume,
63433|     ULONG              SnapShotSequenceNumber,
63434|     ULARGE_INTEGER     LastGranuleFinished );
63435|
63436| //-----
        | -----
        | -
63437|
63438| NTSTATUS CreateMasterListForSequence (
63439|     ULONG              masterSequence,
63440|     LARGE_INTEGER      masterTime,      // time
        | snapshot was created
63441|     pkSnapShotMaster   *&masterList,
63442|     ULONG              &numMastersInList );

```

```

63443|
63444| NTSTATUS GetSequenceForMaster (
63445|     pkSnapshotMaster  master,
63446|     ULONG              &snapshotSequence );
63447|
63448| void LogRevertEvent (
63449|     ULONG    message,
63450|     NTSTATUS status,
63451|     WCHAR    *strings[],
63452|     ULONG    numStrings );
63453|
63454| //-----
| -----
| -
63455|
63456| STATIC void CrashTestDummy()
63457| {
63458|     #ifdef DEBUG
63459|         RevertDebug(("***** CrashTestDummy
| *****\n"));
63460|         int *BadPointer = 0;
63461|         ++(*BadPointer);
63462|     #endif /*DEBUG*/
63463| }
63464|
63465| //-----
| -----
| -
63466|
63467| STATIC NTSTATUS ValidateDirectAccessFile (
| DirectAccessFile *direct, const char *whichOne )
63468| {
63469|     NTSTATUS status = STATUS_SUCCESS;
63470|
63471|     __try {
63472|         if ( direct == NULL ) {
63473|             RevertDebug(("!!! ValidateDirectAccessFile:
| %s DirectAccessFile is NULL !!!\n", whichOne));
63474|             status = STATUS_INVALID_PARAMETER;
63475|         } else {
63476|             if ( !direct->readyForDirectIo() ) {
63477|                 RevertDebug(("!!!
| ValidateDirectAccessFile: %s DirectAccessFile is not
| ready for direct I/O !!!\n", whichOne));
63478|                 status = STATUS_INVALID_PARAMETER;
63479|             }
63480|         }
63481|     } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
63482|         status = GetExceptionCode();

```

```

63483|     RevertDebug(("!!! ValidateDirectAccessFile:
| Exception %08x\n",status));
63484| }
63485|
63486| ASSERT (status == STATUS_SUCCESS);
63487| return status;
63488| }
63489|
63490| //-----
| -----
| -
63491|
63492| STATIC NTSTATUS RevertGranule (
63493|     PFILTERED_EXTENSION    DevExt,
63494|     LARGE_INTEGER          ByteOffset,
63495|     const char              *GranuleBuffer )
63496| {
63497|     bool granuleWasSplitUp = false;
63498|
63499|     NTSTATUS status = ValidateDirectAccessFile
| (DevExt->Cache.CacheFile.Direct, "cache");
63500|     if ( NT_SUCCESS(status) ) {
63501|         status = ValidateDirectAccessFile
| (DevExt->Cache.HeaderFile.Direct, "header");
63502|         if ( NT_SUCCESS(status) ) {
63503|             status = ValidateDirectAccessFile
| (DevExt->Cache.IndexFile.Direct, "index");
63504|         }
63505|     }
63506|
63507|     if ( !NT_SUCCESS(status) ) {
63508|         RevertDebug(("!!! RevertGranule bailing out
| early due to invalid DirectAccessFile object\n"));
63509|         return status;
63510|     }
63511|
63512|     /*CrashTestDummy();*/ //Don says: recovered
| gracefully when tested with exception caused here
63513|
63514|     // Go through each cluster in the granule and see
| if it overlaps with a PSM file.
63515|
63516|     ULONG ClusterSizeInBytes =
| DevExt->Cache.CacheFile.Direct->getClusterSizeInBytes();
63517|     ASSERT ( ClusterSizeInBytes > 0 );
63518|     ASSERT ( GRANULE_SIZE % ClusterSizeInBytes == 0 );
63519|     ASSERT ( GRANULE_SIZE >= ClusterSizeInBytes );
63520|     ULONG ClustersPerGranule = GRANULE_SIZE /
| ClusterSizeInBytes;
63521|     LARGE_INTEGER ByteOffsetInGranule = ByteOffset;

```



```

    | mapStatus==STATUS_SUCCESS );
63562|         }
63563|     }
63564| } __finally {
63565|     pmReleaseReaderLock (
    | &DevExt->Cache.DirectAccessResource );
63566| }
63567|
63568|     if ( mapStatus == STATUS_SUCCESS ) {
63569|         // The cluster belongs to one of the live
    | PSM files.
63570|         RevertDebug(("RevertGranule: *** Found live
    | cluster within granule (disk offset = %016l64x)!
    | ***\n",ByteOffsetInGranule.QuadPart));
63571|         if ( !granuleWasSplitUp ) {
63572|             granuleWasSplitUp = true;
63573|             if ( ci > 0 ) {
63574|                 // We didn't realize before now
    | that this granule needed to be split up.
63575|                 // Therefore, we need to write the
    | front part of the granule.
63576|                 status = Sblo_WriteDevice (
63577|                     DevExt->DeviceObject,
63578|                     &ByteOffset,
63579|                     ci * ClusterSizeInBytes,
63580|                     GranuleBuffer );
63581|
63582|                 RevertDebug(("RevertGranule: ***
    | Wrote first %08x cluster(s) of granule\n",ci));
63583|
63584|                 #ifdef DEBUG
63585|                     NumClustersWritten = ci;
63586|                 #endif /*DEBUG*/
63587|             }
63588|         }
63589|
63590|         #ifdef DEBUG
63591|             ++NumClustersSkipped;
63592|             RevertDebug(("RevertGranule: ***
    | Skipped cluster index %08x in granule\n",ci));
63593|             #endif /*DEBUG*/
63594|     } else {
63595|         ASSERT (mapStatus == STATUS_NOT_FOUND);
63596|         if ( granuleWasSplitUp ) {
63597|             // Write just this cluster...
63598|             status = Sblo_WriteDevice (
63599|                 DevExt->DeviceObject,
63600|                 &ByteOffsetInGranule,
63601|                 ClusterSizeInBytes,
63602|                 GranuleBuffer +

```

```

    | (ULONG)(ByteOffsetInGranule.QuadPart -
    | ByteOffset.QuadPart) );
63603|
63604|         #ifdef DEBUG
63605|             ++NumClustersWritten;
63606|             RevertDebug(("RevertGranule: ***
    | Wrote cluster index %08x in granule\n",ci));
63607|         #endif /*DEBUG*/
63608|     }
63609| }
63610|
63611|     if ( !NT_SUCCESS(status) ) {
63612|         RevertDebug(("RevertGranule: !!!!!!!
    | breaking out of cluster loop early due to
    | %08x\n",status));
63613|         break;
63614|     }
63615|
63616|     ByteOffsetInGranule.QuadPart +=
    | ClusterSizeInBytes;
63617| }
63618|
63619| #ifdef DEBUG
63620|     if ( granuleWasSplitUp ) {
63621|         RevertDebug(("*** Granule split into
    | clusters: wrote %08x, skipped %08x, total %08x\n",
63622|             NumClustersWritten,
63623|             NumClustersSkipped,
63624|             NumClustersWritten +
    | NumClustersSkipped));
63625|
63626|         RevertDebug(("*** Cluster Size = %08x,
    | Granule Size = %08x, Clusters Per Granule = %08x\n",
63627|             ClusterSizeInBytes,
63628|             GRANULE_SIZE,
63629|             ClustersPerGranule));
63630|
63631|         ++TotalGranulesSplit;
63632|         RevertDebug(("*** Total granules split
    | since boot = %08x\n",TotalGranulesSplit));
63633|
63634|         ASSERT ( ClustersPerGranule ==
    | NumClustersWritten + NumClustersSkipped ); //each
    | cluster should either have been skipped or written
63635|         ASSERT ( NumClustersWritten > 0 );
    | //skipping all clusters means the granule should not
    | have been PSM'ed in the first place!!!
63636|         ASSERT ( NumClustersSkipped > 0 ); //if we
    | didn't skip any, then why did we think we split up the
    | granule, hmmm???

```

```

63637|     }
63638| #endif /*DEBUG*/
63639|
63640| if ( !granuleWasSplitUp && NT_SUCCESS(status) ) {
63641|     status = Sblo_WriteDevice (
63642|         DevExt->DeviceObject,
63643|         &ByteOffset,
63644|         GRANULE_SIZE,
63645|         GranuleBuffer );
63646| }
63647|
63648| if ( !NT_SUCCESS(status) ) {
63649|     RevertDebug(("!!!!!! RevertGranule returning
63650|         | %08x - your drive is probably unhappy now\n",status));
63651|     ASSERT(FALSE);
63652| }
63653| return status;
63654| }
63655|
63656|
63657| //-----
63658| | -----
63659| | -
63660| // FinalizeRevert was made a separate function so that
63661| | we could perform the same functions either after
63662| // reverting (if not a system drive) or deferring them
63663| | until after boot (if system volume).
63664|
63665| STATIC void FinalizeRevert ( NTSTATUS status )
63666| {
63667|     RevertDebug(("FinalizeRevert: status passed in is
63668|         | %08x\n",status));
63669|     UpdateLastRevertResult (status);
63670|     UpdateRevertStatus (PSM_IDLE);
63671|     if ( NT_SUCCESS(status) ) {
63672|         LogRevertEvent ( PSM_REVERT_COMPLETE, status,
63673|             | NULL, 0 );
63674|     } else {
63675|         LogRevertEvent ( PSM_REVERT_FAILED, status,
63676|             | NULL, 0 );
63677|     }
63678| }
63679|
63680| //-----
63681| | -----
63682| | -
63683|
63684| STATIC NTSTATUS PerformRevertOperation (
63685|     VolumeEntry     &volume,

```

```

63677|  ULONG          revertFlags,
63678|  unsigned __int64  startingGranule,    //
        | nonzero only during recovery
63679|  bool             preTestFlag )      // if
        | true, does the revert.  if false, just tests cache
        | usage
63680| {
63681|  // For each granule that has changed since the
        | snapshot was taken,
63682|  // copy from the snapshot to the live volume.
63683|
63684|  RevertDebug(("Entering
        | PerformRevertOperation%s\n", (preTestFlag?" (CACHE TEST
        | ONLY)":"")));
63685|  volume.cacheSlotsUsed.QuadPart = 0;
63686|
63687|  WCHAR *eventLogStrings[5] = {0};
63688|
63689|  if ( !(revertFlags & PSM_REVERT_FLAG_IN_PROGRESS) )
        | {
63690|      // Enforce sanity: don't allow starting revert
        | in the middle
63691|      // unless we are explicitly told this is being
        | done in recovery.
63692|      startingGranule = 0;
63693|  }
63694|
63695|  NTSTATUS status = STATUS_SUCCESS;
63696|  PFILTERED_EXTENSION devExt = 0;
63697|  pkSnapshotEntry p = 0;
63698|  PVOID granuleBuffer = (PVOID)
        | MemAllocatePoolWithTag ( PagedPool, GRANULE_SIZE,
        | PSM_REVERT_BUFFER_TAG );
63699|  if ( granuleBuffer ) {
63700|      ULARGE_INTEGER startingSector = {0};
63701|      ULARGE_INTEGER dataSize = {0};
63702|      ULARGE_INTEGER numVolumeSectors = {0};
63703|
63704|      p = volume.snapshot;
63705|      devExt = GetFilteredExtension
        | (p->DeviceObject);
63706|      ULONG SectorSize = devExt->BytesPerSector;
63707|      numVolumeSectors.QuadPart =
        | devExt->Pi.PartitionLength.QuadPart / SectorSize;
63708|
63709|      dataSize.QuadPart = GRANULE_SIZE;
63710|
63711|      ULONG treeSearchFlags = 0;
63712|      if (
        | pPersistentDictionary(p->Dictionary)->IsReadWrite() ) {

```



```

63713|         // If virtual volume is read-write, then we
63714|         // want to revert modifications to snapshot
        | as well.
63715|         treeSearchFlags |= DICT_FLAG_VIRTUAL_IO;
63716|         RevertDebug(("PerformRevertOperation:
        | snapshot is not read-only; including virtual
        | writes\n"));
63717|     }
63718|
63719|     RevertDebug((
63720|         "PerformRevertOperation: starting revert%s
        | of volume=%08x, numSectors=0x%016l64x, ss=%08x\n",
63721|         (preTestFlag ? " test" : ""),
63722|         p->DeviceObject,
63723|         numVolumeSectors.QuadPart,
63724|         p));
63725|
63726|     RevertDebug(("PerformRevertOperation:
        | treeSearchFlags = %08x\n",treeSearchFlags));
63727|
63728|     eventLogStrings[0] = devExt->Name;
63729|
63730|     if ( !volume.isSystemVolume ) {
63731|         UpdateRevertStatus
        | (PSM_REVERT_IN_PROGRESS);
63732|         if ( !preTestFlag ) {
63733|             LogRevertEvent (
63734|                 PSM_REVERT_STARTING_VOLUME,
63735|                 PSM_REVERT_STARTING_VOLUME,
63736|                 eventLogStrings,
63737|                 1 );
63738|         }
63739|     }
63740|
63741|     ULARGE_INTEGER lastGranuleFinished = {0};
63742|     lastGranuleFinished.QuadPart = startingGranule;
63743|
63744|     if ( !preTestFlag ) {
63745|         // Don't commit to recovery of revert
        | before deciding whether we
63746|         // really want to do it!
63747|
63748|         UpdateRevertRecoveryInfo (
63749|             p->DeviceObject,
63750|             p->Dictionary->GetSequenceNumber(),
63751|             lastGranuleFinished );
63752|     }
63753|
63754|     const __int64 RECOVERY_UPDATE_INTERVAL =
        | SECONDS(10);

```

```

63755|     LARGE_INTEGER nextRecoveryUpdateTime = {0};
63756|     KeQuerySystemTime (&nextRecoveryUpdateTime);
63757|     nextRecoveryUpdateTime.QuadPart +=
        | RECOVERY_UPDATE_INTERVAL;
63758|
63759|     for ( startingSector.QuadPart = startingGranule
        | * SECTORS_PER_GRANULE;
63760|         startingSector.QuadPart <
        | numVolumeSectors.QuadPart;
63761|         startingSector.QuadPart +=
        | SECTORS_PER_GRANULE ) {
63762|
63763|         ULONG countDid = 0;
63764|
63765|         NTSTATUS searchStatus =
63766|             p->Dictionary->searchMultiple (
63767|                 devExt,
        | //PFILTERED_EXTENSION DevExt
63768|                 startingSector,
        | //ULARGE_INTEGER StartingSector
63769|                 SECTORS_PER_GRANULE, //ULONG
        | Count
63770|                 countDid,           //ULONG
        | &CountDid
63771|                 NULL,
        | //PRTL_BITMAP
63772|                 dataSize,
        | //ULARGE_INTEGER DataSize
63773|                 granuleBuffer,      //PVOID
        | VirtualDataPointer
63774|                 treeSearchFlags ); //ULONG
        | Flags
63775|
63776|         if ( searchStatus==STATUS_SUCCESS &&
        | countDid==SECTORS_PER_GRANULE ) {
63777|             ++(volume.cacheSlotsUsed.QuadPart);
        | // assume worst case: each granule written uses more
        | cache
63778|             if ( !preTestFlag ) {
63779|                 // Put the cached data granule back
        | onto the original drive.
63780|
63781|                 LARGE_INTEGER byteOffset;
63782|                 byteOffset.QuadPart =
        | startingSector.QuadPart * SectorSize;
63783|
63784|                 RevertDebug((
63785|                     "PerformRevertOperation:
        | reverting volume=%08x ss=%08x startSector=%l64u\n",
63786|                     p->DeviceObject,

```

```

63787|                p,
63788|                startingSector.QuadPart));
63789|
63790|            __try {
63791|                status = RevertGranule (
63792|                    | devExt, byteOffset, (const char *)granuleBuffer );
63793|            } __except(
63794|                | ExceptionFilter(GetExceptionInformation()) ) {
63795|                status = GetExceptionCode();
63796|                RevertDebug(("!!!
63797|                | PerformRevertOperation: Exception %08x in
63798|                | RevertGranule\n",status));
63799|            }
63800|
63801|            if ( status == STATUS_SUCCESS ) {
63802|                // Check to see if this would
63803|                | be a good time to update recovery info
63804|                LARGE_INTEGER currentTime =
63805|                | {0};
63806|                KeQuerySystemTime
63807|                | (&currentTime);
63808|                if ( currentTime.QuadPart >=
63809|                | nextRecoveryUpdateTime.QuadPart ) {
63810|
63811|                | lastGranuleFinished.QuadPart =
63812|                | startingSector.QuadPart
63813|                | / SECTORS_PER_GRANULE;
63814|
63815|                UpdateRevertRecoveryInfo (
63816|                | p->DeviceObject,
63817|                | p->Dictionary->GetSequenceNumber(),
63818|                | lastGranuleFinished );
63819|
63820|                | nextRecoveryUpdateTime.QuadPart =
63821|                | currentTime.QuadPart +
63822|                | RECOVERY_UPDATE_INTERVAL;
63823|            }
63824|        } else {
63825|            RevertDebug(("!!!
63826|            | PerformRevertOperation: RevertGranule returned 0x%08x
63827|            | (ss=%08x
63828|            | sector=%l64u)\n",status,p,startingSector.QuadPart));
63829|            break; // This is really
63830|            | bad... we are in the middle of reverting and have
63831|            | probably screwed up the volume
63832|        }
63833|    }
63834|}
63835|}

```

```

63819|     }
63820|
63821|     FREE_POINTER (granuleBuffer);
63822| } else {
63823|     RevertDebug(("PerformRevertOperation: Out of
        | memory (granuleBuffer)\n"));
63824|     status = STATUS_INSUFFICIENT_RESOURCES;
63825| }
63826|
63827| if ( !volume.isSystemVolume ) {
63828|     UpdateRevertStatus (PSM_IDLE);
63829| }
63830|
63831| if ( !preTestFlag ) {
63832|     // Only do the finalize (registry write and
        | logged event) if this is NOT the system drive
63833|     if ( volume.isSystemVolume ) {
63834|         RevertDebug(("PerformRevertOperation
        | (SysRevert): Skipping FinalizeRevert.\n"));
63835|     } else {
63836|         FinalizeRevert (status);
63837|     }
63838|
63839|     //
63840|     // We call CancelRevertAtBootForVolume when we
        | complete any revert operation
63841|     // so that we won't attempt to do the same
        | revert again during the next boot.
63842|     // If we never get here (say someone hits the
        | computer's reset button) then the next
63843|     // boot will automatically continue reverting
        | at the volume and granule
63844|     // indicated by the last call to
        | UpdateRevertRecoveryInfo.
63845|     //
63846|     if ( NT_SUCCESS(status) ) {
63847|         CancelRevertAtBootForVolume (&volume,
        | p->DeviceObject);
63848|     } else {
63849|         RevertDebug(("PerformRevertOperation:
        | Revert failed due to status=%08x. Will try revert
        | again at next boot.\n",status));
63850|     }
63851|
63852|     #if 0
63853|     {
63854|         // For debugging deadlock issues, we
        | artificially delay in debug builds...
63855|
63856|         const int SecondsToWait = 30;

```

```

63857|         RevertDebug(("@$@$@$@$ Start Post-Revert
| Test Delay (%d seconds) @$@$@$@$\\n",SecondsToWait));
63858|
63859|         LARGE_INTEGER TimeToWait;
63860|         TimeToWait.QuadPart =
| RELATIVE(SECONDS(SecondsToWait));
63861|         KeDelayExecutionThread(
63862|             (KPROCESSOR_MODE)KernelMode, // IN
| KPROCESSOR_MODE WaitMode,
63863|             FALSE, // IN BOOLEAN
| Alertable,
63864|             &TimeToWait ); // IN PLARGE_INTEGER
| Interval
63865|
63866|         RevertDebug(("@$@$@$@$ End Post-Revert
| Test Delay @$@$@$@$\\n");
63867|     }
63868|     #endif /*DEBUG*/
63869| }
63870|
63871| RevertDebug(("PerformRevertOperation%s returning
| %08x\\n",(preTestFlag ? " (CACHE TEST
| ONLY)": ""),status));
63872| return status;
63873| }
63874|
63875| //-----
| -----
| -
63876|
63877| STATIC NTSTATUS RemountAffectedVolumes (
63878|     VolumeEntry *volumeList,
63879|     ULONG numVolumes,
63880|     ULONG revertFlags )
63881| {
63882|     RevertDebug(("Entering RemountAffectedVolumes;
| numVolumes=%u,
| volumeList=%08x\\n",numVolumes,volumeList));
63883|     NTSTATUS status = STATUS_SUCCESS;
63884|
63885|     #ifdef DEBUG
63886|         #if 0
63887|             // Delay for testing purposes... this way I can
| check to see if mount
63888|             // request for the volume(s) really fails
63889|             RevertDebug(("@$@$@$@$ Start Pre-Mount Test
| Delay @$@$@$@$\\n");
63890|
63891|             LARGE_INTEGER TimeToWait;
63892|             TimeToWait.QuadPart = RELATIVE(SECONDS(2*60));

```

```

63893|     KeDelayExecutionThread(
63894|         (KPROCESSOR_MODE)KernelMode, // IN
        | KPROCESSOR_MODE WaitMode,
63895|         FALSE, // IN BOOLEAN Alertable,
63896|         &TimeToWait ); // IN PLARGE_INTEGER
        | Interval
63897|
63898|     RevertDebug(("@$@$@$@$ End Pre-Mount Test
        | Delay @$@$@$@$\\n"));
63899| #endif
63900| #endif /*DEBUG*/
63901|
63902|     __try {
63903|         // Unlock and remount all involved volumes.
63904|         for ( ULONG volumeIndex=0;
            | volumeIndex<numVolumes; ++volumeIndex ) {
63905|             VolumeEntry &thisEntry =
            | volumeList[volumeIndex];
63906|             //pkSnapshotEntry p = thisEntry.snapshot;
63907|             PFILTERED_EXTENSION devExt =
            | GetFilteredExtension
            | (volumeList[volumeIndex].volumeObject);
63908|
63909|             devExt->IsReverting = FALSE; //
            | otherwise remount will be failed by filesystem filter
63910|
63911|             if ( thisEntry.locked ) {
63912|                 PFILE_OBJECT VolumeFileObject = NULL;
63913|                 HANDLE VolumeHandle =
            | INVALID_HANDLE_VALUE;
63914|                 NTSTATUS openStatus =
            | Sblo_OpenVolumeHandle (devExt->Name, VolumeHandle,
            | VolumeFileObject, 0);
63915|                 if ( NT_SUCCESS(openStatus) ) {
63916|                     NTSTATUS unlockStatus =
            | FS_UnlockVolume (VolumeFileObject);
63917|                     if ( NT_SUCCESS(unlockStatus) ) {
63918|                         thisEntry.locked = 0;
63919|
            | RevertDebug(("RemountAffectedVolumes: volume '%S'
            | unlocked.\\n",devExt->Name));
63920|                     } else {
63921|
            | RevertDebug(("RemountAffectedVolumes: !!! Error %08x
            | unlocking volume '%S\\n",unlockStatus,devExt->Name));
63922|                     }
63923|                     Sblo_CloseVolumeHandle
            | (VolumeHandle, VolumeFileObject);
63924|                     } else {
63925|

```

```

    | RevertDebug(("RemountAffectedVolumes: !!! Error %08x
    | opening volume '%S' for
    | unlock\n",openStatus,devExt->Name));
63926|         }
63927|     }
63928|
63929|     if ( thisEntry.dismounted ) {
63930|         SbTouchVolume (devExt->Name);
63931|         thisEntry.dismounted = 0;
63932|         RevertDebug(("RemountAffectedVolumes:
    | volume '%S' remounted.\n",devExt->Name));
63933|     } else if ( thisEntry.dismountFailed ) {
63934|         RevertDebug(("RemountAffectedVolumes:
    | volume '%S' previously failed dismount - starting
    | part2\n",devExt->Name));
63935|
63936|         HANDLE ThreadHandle =
    | INVALID_HANDLE_VALUE;
63937|
    | pmStartThread(PersistentDictionary::Part2OfRebuildForVol
    | ume,devExt->DeviceObject,&ThreadHandle);
63938|
    | ZwWaitForSingleObject(ThreadHandle,FALSE,NULL);
63939|         ZwClose(ThreadHandle);
63940|
63941|         RevertDebug(("RemountAffectedVolumes:
    | after waiting for part2 thread to finish on
    | '%S'\n",devExt->Name));
63942|     }
63943| }
63944| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
63945|     status=GetExceptionCode();
63946|     RevertDebug(("RemountAffectedVolumes: Error!
    | Exception %08x\n",status));
63947| }
63948|
63949| RevertDebug(("RemountAffectedVolumes returning
    | %08x\n",status));
63950| return status;
63951| }
63952|
63953| //-----
    | -----
    | -
63954|
63955| STATIC NTSTATUS DismountVolumeEntry ( VolumeEntry
    | &volumeEntry )
63956| {
63957|     NTSTATUS status = STATUS_SUCCESS;

```

```

63958|   PFILTERED_EXTENSION devExt = GetFilteredExtension
        | (volumeEntry.volumeObject);
63959|
63960|   __try {
63961|       PFILE_OBJECT VolumeFileObject = NULL;
63962|       HANDLE VolumeHandle = INVALID_HANDLE_VALUE;
63963|
63964|       ASSERT ( !volumeEntry.dismounted );
63965|       ASSERT ( !volumeEntry.locked );
63966|
63967|       status = Sblo_OpenVolumeHandle (devExt->Name,
        | VolumeHandle, VolumeFileObject, GENERIC_READ);
63968|
63969|       if ( NT_SUCCESS(status) ) {
63970|           ASSERT ( VolumeFileObject != NULL );
63971|           ASSERT ( IsValidHandle(VolumeHandle) );
63972|           status = FS_DismountVolume (
        | VolumeFileObject );
63973|           RevertDebug(("DismountVolumeEntry:
        | FS_DismountVolume returned %08x for
        | '%S'\n",status,devExt->Name));
63974|           if ( NT_SUCCESS(status) ) {
63975|               volumeEntry.dismounted = 1;
63976|           }
63977|           Sblo_CloseVolumeHandle ( VolumeHandle,
        | VolumeFileObject );
63978|       }
63979|   } __except(
        | ExceptionFilter(GetExceptionInformation()) ) {
63980|       status = GetExceptionCode();
63981|       RevertDebug(("!!! Exception %08x in
        | DismountVolumeEntry\n",status));
63982|   }
63983|
63984|   if ( !NT_SUCCESS(status) ) {
63985|       RevertDebug(("DismountVolumeEntry: Dismount
        | failed for '%S'\n", devExt->Name));
63986|       volumeEntry.dismountFailed = 1; // set flag
        | so later we know we need to run part2 to reload
        | snapshots.
63987|   }
63988|
63989|   return status;
63990| }
63991|
63992| //-----
        | -----
        | -
63993|
63994| NTSTATUS DismountVolumeList (

```



```

63995| VolumeEntry    *volumeList,
63996| ULONG          numVolumes )
63997| {
63998|     NTSTATUS status = 0;
63999|     ULONG dismountedCount = 0;
64000|
64001|     for ( ULONG volumeIndex=0; volumeIndex<numVolumes;
        | ++volumeIndex ) {
64002|         status = DismountVolumeEntry
        | (volumeList[volumeIndex]);
64003|         if ( NT_SUCCESS(status) ) {
64004|             ++dismountedCount;
64005|         } else {
64006|             break;
64007|         }
64008|     }
64009|
64010|     RevertDebug(("DismountVolumeList: dismounted %u
        | volume%s; status=%08x\n",
64011|         dismountedCount,
64012|         dismountedCount==1 ? "" : "s",
64013|         status ));
64014|
64015|     return status;
64016| }
64017|
64018| //-----
        | -----
        | -
64019|
64020| STATIC NTSTATUS PrepareVolumeList (
64021|     pkSnapShotMaster  master,
64022|     PDEVICE_OBJECT    volumeDeviceObject,
64023|     VolumeEntry        *volumeList,
64024|     const ULONG        maxNumVolumes,
64025|     ULONG              &numVolumes,
64026|     ULONG              enableSystemVolumeRevert )
64027| {
64028|     RevertDebug((
64029|         "Entering PrepareVolumeList; master=%08x,
        | volDevObj=%08x, volumeList=%08x\n",
64030|         master,
64031|         volumeDeviceObject,
64032|         volumeList));
64033|
64034|     NTSTATUS status = STATUS_SUCCESS;
64035|     numVolumes = 0;
64036|     pkSnapShotEntry p = 0;
64037|     GetSnapShotForRead();
64038|     __try {

```

```

64039|     p = GetTopSnapShotForMaster (
        | &master->SnapShots );
64040|     while ( p ) {
64041|
        | //-----
        | -----
64042|         // If volumeDeviceObject==NULL, it means
        | that we want to include every volume
64043|         // in the snapshot. Otherwise, it is
        | interpreted to be the specific device object
64044|         // belonging to the single volume to do.
64045|         //
64046|         if ( volumeDeviceObject==NULL ||
        | p->DeviceObject==volumeDeviceObject ) {
64047|
        | //-----
64048|         // Make sure we don't overflow the
        | array...
64049|         //
64050|         if ( numVolumes >= maxNumVolumes ) {
64051|             RevertDebug(("PrepareVolumeList
        | (revert): More than %u volumes found!\n",
        | maxNumVolumes));
64052|
64053|             // fail the snapshot
64054|             DoneWithSnapShot(p); // because
        | we called either GetTopSnapShotForMaster or
        | GetNextSnapShotForMaster
64055|             status =
        | STATUS_INSUFFICIENT_RESOURCES;
64056|             break;
64057|         }
64058|
64059|         RevertDebug(("PrepareVolumeList:
        | list[%d] = ss %08x\n",numVolumes,p));
64060|         VolumeEntry &thisEntry =
        | volumeList[numVolumes++];
64061|         UseSnapShot(p); // increment
        | reference counter until we are done with ss
64062|         PFILTERED_EXTENSION devExt =
        | GetFilteredExtension ( p->DeviceObject );
64063|
64064|         thisEntry.snapshot      = p;
64065|         thisEntry.volumeld      =
        | devExt->Volumeld;
64066|         thisEntry.dismounted    = 0;
64067|         thisEntry.locked        = 0;
64068|         thisEntry.acquired      = 1;
64069|         thisEntry.dismountFailed = 0;
64070|         thisEntry.isSystemVolume =

```

```

        | IsSystemVolume(devExt->Name) ? 1 : 0;
64071|         thisEntry.volumeObject    =
        | p->DeviceObject;
64072|
64073|
        | //-----
        | -----
64074|         //  Check for enabled/disabled system
        | drive revert...
64075|         //
64076|         if ( !enableSystemVolumeRevert &&
        | thisEntry.isSystemVolume ) {
64077|
        | //-----
        | -----
64078|         //  This means the single- or
        | multi-volume snapshot contains the system drive,
64079|         //  but reverting the system drive
        | is not allowed.  We fail it now.
64080|
64081|         ASSERT(numVolumes>0 &&
        | numVolumes<=maxNumVolumes); // we must have added at
        | least one volume to the list.
64082|         --numVolumes; // this erases the
        | current entry 'thisEntry' (i.e. the last entry) in the
        | list.
64083|         RevertDebug(("(Revert)
        | PrepareVolumeList: Failing snapshot that contains the
        | system volume!\n"));
64084|
64085|         // fail the snapshot
64086|         DoneWithSnapShot(p); // call once
        | for UseSnapShot() and ...
64087|         DoneWithSnapShot(p); // ... twice
        | for Get[Top|Next]SnapShotForMaster.
64088|         status = PSM_REVERT_FAILED; //
        | FIXFIXFIX: When we can re-localize, add new error for
        | "System Volume Revert Disabled".
64089|         break;
64090|     }
64091|
64092|     devExt->IsReverting = TRUE; //
        | tells filesystem filter to prevent mounts and snapshot
        | reloads
64093|     }
64094|
64095|     p =
        | GetNextSnapShotForMaster(&master->SnapShots,p);
64096|     }
64097| } __finally {

```

```

64098|     ReleaseSnapShotForRead();
64099| }
64100|
64101| if ( numVolumes == 0 ) {
64102|     if ( NT_SUCCESS(status) ) {
64103|         // This is here to make sure we return an
        | error if we have no volumes in the snapshot.
64104|         // We need to return an error code to make
        | sure the failure gets event logged.
64105|         status = STATUS_NOT_FOUND;
64106|     }
64107|     RevertDebug(("PrepareVolumeList (revert): Could
        | not find any volumes for master!\n"));
64108| } else {
64109|     if ( volumeDeviceObject != NULL ) {
64110|         // If a particular volume was specified, no
        | more than one snapshot
64111|         // should have been found!
64112|         ASSERT (numVolumes == 1);
64113|     }
64114| }
64115|
64116| RevertDebug(("PrepareVolumeList returning
        | %08x\n",status));
64117| return status;
64118| }
64119|
64120| //-----
        | -----
        | -
64121|
64122| STATIC ULONG CalcDeviceNamesLengthInBytes (
64123|     VolumeEntry *volumeList,
64124|     ULONG numVolumes )
64125| {
64126|     ULONG numChars = 1;    // account for extra '\0'
        | at end of list
64127|
64128|     for ( ULONG i=0; i<numVolumes; ++i ) {
64129|         PFILTERED_EXTENSION devExt =
        | GetFilteredExtension (volumeList[i].volumeObject);
64130|         numChars += wcslen(devExt->Name) + 1;
64131|     }
64132|
64133|     return numChars * sizeof(WCHAR);
64134| }
64135|
64136| //-----
        | -----
        | -

```

```

64137|
64138| NTSTATUS GetSequenceForMaster (
64139|     pkSnapshotMaster  master,
64140|     ULONG              &snapshotSequence )
64141| {
64142|     NTSTATUS status = STATUS_NOT_FOUND;
64143|     snapshotSequence = 0;
64144|
64145|     if ( !master ) {
64146|         RevertDebug(("GetSequenceForMaster: NULL
        | master encountered!\n"));
64147|         status = STATUS_INVALID_PARAMETER;
64148|     } else {
64149|         GetSnapshotForRead();
64150|         __try {
64151|             PDEVICE_OBJECT DevObj =
        | PSMAN_DRIVER_OBJECT->DeviceObject;
64152|             while(DevObj) {
64153|
        | if(PsmGetObjectTypes(DevObj)==OBJECT_FILTERED_DISK) {
64154|                 PFILTERED_EXTENSION DevExt =
        | GetFilteredExtension(DevObj);
64155|                 pkSnapshotEntry p = GetTopSnapshot
        | ( &(DevExt->Snapshots) );
64156|                 while ( p ) {
64157|                     if ( p->Dictionary ) {
64158|                         if ( master ==
        | p->MasterSnapshot ) {
64159|                             snapshotSequence =
        | p->Dictionary->GetSequenceNumber();
64160|
        | RevertDebug(("GetSequenceForMaster: found
        | sequence=%08x from
        | snapshot=%08x\n",snapshotSequence,p));
64161|                             DoneWithSnapshot(p);
64162|                             status =
        | STATUS_SUCCESS;
64163|                             break;
64164|                         }
64165|                     }
64166|
        | p =
        | GetNextSnapshot(&(DevExt->Snapshots),p);
64168|                 }
64169|             }
64170|
        | if ( status == STATUS_SUCCESS ) {
64171|                 break;
64172|             }
64173|         }
64174|

```

```

64175|         DevObj = DevObj->NextDevice;
64176|     }
64177| } __finally {
64178|     ReleaseSnapShotForRead();
64179| }
64180| }
64181|
64182| RevertDebug(("GetSequenceForMaster: status=%08x,
| in master=%08x, out seq=%08x\n",
64183|     status,
64184|     master,
64185|     snapShotSequence));
64186|
64187| return status;
64188| }
64189|
64190| //-----
| -----
| -
64191|
64192| NTSTATUS GetMasterForSequenceAndVolume (
64193|     ULONG             snapShotSequence,
64194|     LARGE_INTEGER     snapShotTime,
64195|     PDEVICE_OBJECT     volumeDeviceObject,
64196|     pkSnapShotMaster   &master )
64197| {
64198|     NTSTATUS status = STATUS_NOT_FOUND;
64199|     master = 0;
64200|     RevertDebug(("GetMasterForSequenceAndVolume:
| sequence=%08x, volumeDeviceObject=%08x\n",
| snapShotSequence, volumeDeviceObject));
64201|
64202|     ASSERT ( volumeDeviceObject != NULL );
64203|     PFILTERED_EXTENSION devExt = GetFilteredExtension
| (volumeDeviceObject);
64204|     ASSERT ( devExt != NULL );
64205|
64206|     GetSnapShotForRead();
64207|     __try {
64208|         pkSnapShotEntry p = GetTopSnapShot (
| &(devExt->SnapShots) );
64209|         while ( p ) {
64210|             if ( p->Dictionary ) {
64211|                 ULONG seq =
| p->Dictionary->GetSequenceNumber();
64212|                 __int64 time =
| p->Dictionary->GetSnapShotTime();
64213|                 if ( time == snapShotTime.QuadPart ) {
64214|                     ASSERT ( p->MasterSnapShot != NULL
| );

```

```

64215|         ASSERT (seq == snapShotSequence);
64216|         if ( p->MasterSnapShot ) {
64217|             master = p->MasterSnapShot;
64218|             status = STATUS_SUCCESS;
64219|         } else {
64220|             RevertDebug(("!!!
| GetMasterForSequenceAndVolume: MasterSnapShot==NULL in
| snapshot entry %08x\n",p));
64221|         }
64222|         DoneWithSnapShot(p);
64223|         break;
64224|     }
64225| }
64226| p = GetNextSnapShot(&(devExt->SnapShots),
| p);
64227| }
64228| } __finally {
64229|     ReleaseSnapShotForRead();
64230| }
64231|
64232| RevertDebug(("GetMasterForSequenceAndVolume
| returning %08x, master=%08x\n",status,master));
64233| return status;
64234| }
64235|
64236| //-----
| -----
| -
64237|
64238| NTSTATUS GetMasterListForSequence (
64239|     ULONG          snapShotSequence,
64240|     LARGE_INTEGER  snapShotTime,
64241|     pkSnapShotMaster masterList[],
64242|     ULONG          masterListMaxEntries,
64243|     ULONG          &numMastersFound )
64244| {
64245|     NTSTATUS status = STATUS_SUCCESS;
64246|     RevertDebug(("GetMasterListForSequence:
| sequence=%08x\n",snapShotSequence));
64247|     numMastersFound = 0;
64248|
64249|     pkSnapShotMaster master = 0;
64250|     GetSnapShotForRead();
64251|     __try {
64252|         PDEVICE_OBJECT DevObj =
| PSMANDriverObject->DeviceObject;
64253|         while(DevObj && NT_SUCCESS(status)) {
64254|             if(PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK) {
64255|                 PFILTERED_EXTENSION DevExt =

```

```

    | GetFilteredExtension(DevObj);
64256|
64257|         pkSnapShotEntry p = GetTopSnapShot (
    | &(DevExt->SnapShots) );
64258|         while ( p ) {
64259|             if ( p->Dictionary ) {
64260|                 ULONG seq =
    | p->Dictionary->GetSequenceNumber();
64261|                 ASSERT ( p->MasterSnapShot !=
    | NULL );
64262|                 if ( p->MasterSnapShot != NULL
    | ) {
64263|                     if (
    | p->MasterSnapShot->SnapShotTime.QuadPart ==
    | snapShotTime.QuadPart ) {
64264|                         bool AlreadyInList =
    | false;
64265|                         for ( ULONG mi=0; mi <
    | numMastersFound; ++mi ) {
64266|                             if ( masterList[mi]
    | == p->MasterSnapShot ) {
64267|                                 AlreadyInList =
    | true;
64268|                                 break;
64269|                             }
64270|                         }
64271|
64272|                         if ( !AlreadyInList ) {
64273|                             ASSERT ( seq ==
    | snapShotSequence );
64274|                             if (
    | numMastersFound >= masterListMaxEntries ) {
64275|                                 status =
    | PSM_ERROR_INSUFFICIENT_BUFFER;
64276|
    | RevertDebug(("GetMasterListForSequence:
    | masterListMaxEntries=%08x is not big
    | enough!\n",masterListMaxEntries));
64277|
    | DoneWithSnapShot(p);
64278|                                 break;
64279|                             }
64280|
64281|
    | RevertDebug(("GetMasterListForSequence: Adding master
    | %08x to
    | masterList[%08x]\n",p->MasterSnapShot,numMastersFound));
64282|
    | masterList[numMastersFound++] = p->MasterSnapShot;
64283|                 }

```



```

64284|             } else {
64285|         | RevertDebug(("GetMasterListForSequence: NOTE: master
        | %08x has same sequence, but time %016l64x does not
        | match!\n",p->MasterSnapShot,p->MasterSnapShot->SnapShotT
        | ime.QuadPart));
64286|             }
64287|         }
64288|     }
64289|     p =
        | GetNextSnapShot(&(DevExt->SnapShots), p);
64290|     }
64291|     }
64292|     DevObj = DevObj->NextDevice;
64293|     }
64294| } __finally {
64295|     ReleaseSnapShotForRead();
64296| }
64297|
64298| if ( NT_SUCCESS(status) ) {
64299|     if ( numMastersFound == 0 ) {
64300|         status = STATUS_NOT_FOUND;
64301|     }
64302| }
64303|
64304| RevertDebug(("GetMasterListForSequence returning
        | %08x, numMastersFound=%08x\n",status,numMastersFound));
64305| return status;
64306| }
64307|
64308| //-----
        | -----
        | -
64309|
64310| NTSTATUS CreateRevertUndoSnapShot (
64311|     VolumeEntry      *volumeList,
64312|     ULONG             numVolumes,
64313|     ULONG             revertFlags,
64314|     ULONG             &revertUndoSequence )
64315| {
64316|     NTSTATUS status = STATUS_SUCCESS;
64317|     ULONG volumeIndex = 0;
64318|     RevertDebug(("Entering CreateRevertUndoSnapShot;
        | numVolumes=%u\n",numVolumes));
64319|     revertUndoSequence = 0;
64320|
64321|     pOT_USER user =
        | FindPSMUser(PsGetCurrentProcess(),(_ETHREAD*)(-2));
64322|     if ( !user ) {
64323|         RevertDebug(("CreateRevertUndoSnapShot:

```

```

    | FindPSMUser() returned NULL!\n"));
64324|     status = PSM_ERROR_UNSUCCESSFUL;
64325| } else {
64326|     const ULONG tempStringChars = 256;
64327|     const ULONG tempStringBytes = sizeof(WCHAR) *
    | tempStringChars;
64328|     WCHAR *tempString = (WCHAR *) MemAllocateString
    | (tempStringChars);
64329|
64330|     if ( tempString ) {
64331|         ULONG DeviceNamesLengthInBytes =
    | CalcDeviceNamesLengthInBytes (
64332|             volumeList,
64333|             numVolumes );
64334|
64335|         const ULONG PointerBytes = (1 + numVolumes)
    | * sizeof(WCHAR*);
64336|         const ULONG ExtraBytesAtEnd = PointerBytes
    | + DeviceNamesLengthInBytes;
64337|         const ULONG OTISize =
    | sizeof(tOpenTransactionInInternal) + ExtraBytesAtEnd;
64338|
64339|         tOpenTransactionInInternal *oti =
    | (tOpenTransactionInInternal *)MemAllocatePoolWithTag (
64340|             PagedPool,
64341|             OTISize, TEMPTAG );
64342|
64343|         const ULONG OTOSize =
64344|             sizeof(tOpenTransactionOutInternal) +
64345|             sizeof(WCHAR*) * numVolumes +
64346|             sizeof(WCHAR) * (100 * numVolumes);
64347|
64348|         tOpenTransactionOutInternal *oto =
    | (tOpenTransactionOutInternal *)MemAllocatePoolWithTag (
64349|             PagedPool,
64350|             OTOSize, TEMPTAG );
64351|
64352|         if ( !oti || !oto ) {
64353|             RevertDebug(("CreateRevertUndoSnapShot:
    | cannot allocate oti/oto\n"));
64354|             status = STATUS_INSUFFICIENT_RESOURCES;
64355|         } else {
64356|             RtlZeroMemory ( oto, OTOSize );
64357|             RtlZeroMemory ( oti, OTISize );
64358|             oti->NumberOfDevices = numVolumes;
64359|
64360|             ULONG NextNameOffset =
    | sizeof(tOpenTransactionInInternal) + PointerBytes;
64361|             WCHAR *NextName = (WCHAR *) ((char*)oti
    | + NextNameOffset);

```

```

64362|
64363|         for ( volumeIndex=0;
        | volumeIndex<numVolumes; ++volumeIndex ) {
64364|             //pkSnapShotEntry p =
        | volumeList[volumeIndex].snapshot;
64365|             PFILTERED_EXTENSION devExt =
        | GetFilteredExtension
        | (volumeList[volumeIndex].volumeObject);
64366|             wcscopy ( NextName, devExt->Name );
64367|             oti->DeviceName[volumeIndex] =
        | NextNameOffset;
64368|             ULONG CharsToSkip = 1 +
        | wcslen(devExt->Name);
64369|             NextName += CharsToSkip;
64370|             NextNameOffset += sizeof(WCHAR) *
        | CharsToSkip;
64371|         }
64372|
64373|         *NextName = 0;
64374|         oti->DeviceName[volumeIndex] = 0;
64375|
64376|         oti->NumToKeep = -1;
64377|         oti->Priority = 254;
64378|         oti->InternalFlags =
        | PSM_IFLAG_PERSISTENT;
64379|         oti->QuiescentTimeout = 5;
64380|         oti->QuiescentWait = 60;
64381|         oti->SnapShotFlags =
        | PSM_SS_FLAG_P_READONLY;
64382|
64383|         tkSnapShotMaster
        | *revertUndoSnapShotMaster = 0;
64384|
64385|         status = WaitForQuiescentPeriod (
64386|             user,
64387|             oti,
64388|             &revertUndoSnapShotMaster );
64389|
64390|         RevertDebug(("CreateRevertUndoSnapShot:
        | WaitForQuiescentPeriod returned %08x\n",status));
64391|         if ( NT_SUCCESS(status) ) {
64392|             RevertDebug(("PSM Opened revert
        | undo snapshot=%08x\n",revertUndoSnapShotMaster));
64393|             InterlockedIncrement((PLONG)
        | &GlobalData->NumActive);
64394|             | RevertDebug(("CreateRevertUndoSnapShot: Incremented
        | NumActive to %08x\n",GlobalData->NumActive));
64395|             PsmActive = TRUE;
64396|             InterlockedIncrement((PLONG)

```

```

    | &user->Open);
64397|
    | LogOpen(oti,revertUndoSnapShotMaster);
64398|
64399|          // map drives and get instance
    | number
64400|          status =
    | VDiskMapInDrives(revertUndoSnapShotMaster,oti,OTOSize,ot
    | o);
64401|
64402|          if(!INT_SUCCESS(status)) {
64403|          // failed, clean up..
64404|
    | RevertDebug(("CreateRevertUndoSnapShot:
    | VdiskMapInDrives Failed %08x for %08x
    | %08x\n",status,user,revertUndoSnapShotMaster));
64405|
    | InternalClosePSM(user,revertUndoSnapShotMaster);
64406|
64407|          } else {
64408|
    | if(OTOSize>=sizeof(tOpenTransactionOutInternal)) {
64409|          // return back pointer to
    | this so they can pass it to use on close
64410|          oto->KernelSnapShotPointer
    | = revertUndoSnapShotMaster;
64411|          oto->SnapShotTime =
    | revertUndoSnapShotMaster->SnapShotTime;
64412|          oto->Instance =
    | revertUndoSnapShotMaster->Instance;
64413|
64414|          GetSnapShotForRead();
64415|          __try {
64416|          pkSnapShotEntry s =
    | GetTopSnapShotForMaster(&revertUndoSnapShotMaster->SnapS
    | hots);
64417|          if(s) {
64418|
    | s->Dictionary->GetOutParams(oto->CacheFileName);
64419|          revertUndoSequence
    | = s->Dictionary->GetSequenceNumber();
64420|
    | DoneWithSnapShot(s);
64421|          }
64422|          } __finally {
64423|
    | ReleaseSnapShotForRead();
64424|          }
64425|          }
64426|          }

```

```

64427|
64428|         for ( volumeIndex=0;
        | volumeIndex<numVolumes; ++volumeIndex ) {
64429|             pkSnapShotEntry p =
        | volumeList[volumeIndex].snapshot;
64430|             // Create snapshot path name
        | for the revert undo snapshot...
64431|
64432|             swprintf ( tempString,
        | L"%s\\revert.%u", gSnapShotDirName, revertUndoSequence
        | );
64433|             NTSTATUS sns =
        | volumeList[volumeIndex].setNameStatus = SbSetUserName (
64434|                 revertUndoSnapShotMaster,
64435|                 tempString,
64436|                 tempStringBytes );
64437|
64438|             if ( !NT_SUCCESS(sns) ) {
64439|
        | RevertDebug(("CreateRevertUndoSnapShot:
        | SetNameStatus=%08x, volumeIndex=%u,
        | pkSnapShotEntry=%08x\n",sns,volumeIndex,p));
64440|             }
64441|         }
64442|     }
64443| }
64444|
64445|     if ( oti ) {
64446|         FREE_POINTER (oti);
64447|     }
64448|
64449|     if ( oto ) {
64450|         FREE_POINTER (oto);
64451|     }
64452|
64453|     MemFreeString(tempString);
64454| } else {
64455|     RevertDebug(("CreateRevertUndoSnapShot:
        | Cannot allocate memory for 'tempString'\n"));
64456|     status = STATUS_INSUFFICIENT_RESOURCES;
64457| }
64458| }
64459|
64460| RevertDebug(("CreateRevertUndoSnapShot returning
        | %08x\n",status));
64461| return status;
64462| }
64463|
64464| //-----
        | -----

```

```

| -
64465|
64466| STATIC void DoneWithRevertVolumes (
64467|     VolumeEntry  *volumeList,
64468|     ULONG         numVolumes,
64469|     bool          revertAtBoot )
64470| {
64471|     GetSnapShotForRead();
64472|     __try {
64473|         for ( ULONG volumeIndex=0;
64474|              | volumeIndex<numVolumes; ++volumeIndex ) {
64475|             pkSnapShotEntry p =
64476|             | volumeList[volumeIndex].snapshot;
64477|             ASSERT ( p != NULL );
64478|             if ( p != NULL ) {
64479|                 // Do not enable remount on the volume
64480|                 | if it is the system volume being reverted at boot...
64481|                 // This is because we are about to halt
64482|                 | the machine and tell the user to reboot.
64483|                 PFILTERED_EXTENSION devExt =
64484|                 | GetFilteredExtension
64485|                 | (volumeList[volumeIndex].volumeObject);
64486|                 if ( revertAtBoot &&
64487|                     | volumeList[volumeIndex].isSystemVolume ) {
64488|                     RevertDebug(("DoneWithRevertVolumes
64489|                     | (SysRevert): Skipping mount enable for '%S'\n",
64490|                     | devExt->Name));
64491|                     //ASSERT(numVolumes == 1); //
64492|                     | confirm that doing revert-at-boot of system drive
64493|                     | reverts only that one drive.
64494|                 } else {
64495|                     RevertDebug(("DoneWithRevertVolumes
64496|                     | (non SysRevert): Enabling remount on
64497|                     | '%S'\n",devExt->Name));
64498|                     devExt->IsReverting = FALSE; //
64499|                     | because sometimes we don't call RemountAffectedVolumes
64500|                 }
64501|
64502|                 if ( volumeList[volumeIndex].acquired )
64503|                 | {
64504|                     DoneWithSnapShot (p);
64505|                     volumeList[volumeIndex].acquired =
64506|                     | 0;
64507|                 }
64508|             }
64509|         }
64510|     } __finally {
64511|         ReleaseSnapShotForRead();
64512|     }

```

```

64498| }
64499|
64500| //-----
64501| | -----
64502| | -
64503| NTSTATUS RevertVolumeAtNextBoot (
64504|     PDEVICE_OBJECT Volume,
64505|     ULONG SnapshotSequenceNumber,
64506|     LARGE_INTEGER SnapshotTime,
64507|     ULONG CancelOnlyRevertFlags )
64508| {
64509|     NTSTATUS status = PSM_ERROR_REBOOT_NEEDED;
64510|     tRevertInfo revertInfo = {0};
64511|     revertInfo.LastKnownGranuleFinished.QuadPart = 0;
64512|     revertInfo.SnapshotSequenceNumber =
64513|         SnapshotSequenceNumber;
64514|     revertInfo.SnapshotTime = SnapshotTime;
64515|     if ( SnapshotSequenceNumber == 0 ) {
64516|         // This is really a "cancel revert at boot"
64517|         // operation.
64518|         // We use CancelOnlyRevertFlags here. It can
64519|         // be either 0 (revert is complete)
64520|         // or PSM_REVERT_FLAG_FINALIZE_REVERT (system
64521|         // drive was reverted, need to log
64522|         // events and update registry RevertStatus on
64523|         // next boot).
64524|         RevertDebug(("RevertVolumeAtNextBoot
64525|             | (SysRevert): Using CancelOnlyRevertFlags\n"));
64526|         revertInfo.RevertFlags = CancelOnlyRevertFlags;
64527|     } else {
64528|         revertInfo.RevertFlags =
64529|             PSM_REVERT_FLAG_CREATE_UNDO_SNAPSHOT;
64530|     }
64531|     RevertDebug(("RevertVolumeAtNextBoot (SysRevert):
64532|         | Setting persistent RevertFlags to %08x\n",
64533|         | revertInfo.RevertFlags));
64534|     PersistentDictionary::SetRevertBootInfo (Volume,
64535|         | revertInfo);
64536|     status = PersistentDictionary::SaveHeader (Volume);
64537|     if ( NT_SUCCESS(status) ) {
64538|         if ( SnapshotSequenceNumber != 0 ) {
64539|             status = PSM_ERROR_REBOOT_NEEDED;
64540|             WCHAR ssSequenceString[32];
64541|             _ltow ( SnapshotSequenceNumber,

```

```

    | ssSequenceString, 16, sizeof(ssSequenceString)-1 );
64536|     WCHAR *eventLogStrings[] = {
    | ssSequenceString };
64537|     LogRevertEvent (
    | PSM_SCHEDULED_REVERT_AT_BOOT, status, eventLogStrings,
    | 1 );
64538|     UpdateRevertStatus
    | (PSM_STATUS_REVERT_AT_BOOT);
64539| }
64540| } else {
64541|     RevertDebug(("RevertVolumeAtNextBoot: Error
    | %08x saving header!\n",status));
64542| }
64543|
64544|     RevertDebug(("RevertVolumeAtNextBoot: in seq=%08x,
    | out status=%08x\n",SnapShotSequenceNumber,status));
64545|     return status;
64546| }
64547|
64548| //-----
    | -----
    | -
64549| // UpdateRevertRecoveryInfo
64550| //
64551| // This function saves information about the state of
    | the revert operation
64552| // in progress every now and then. If the computer
    | gets rebooted in the
64553| // middle of a revert in progress, the revert boot
    | check will notice and
64554| // continue where the interrupted revert left off.
64555|
64556| NTSTATUS UpdateRevertRecoveryInfo (
64557|     PDEVICE_OBJECT    Volume,
64558|     ULONG              SnapShotSequenceNumber,
64559|     ULARGE_INTEGER     LastGranuleFinished )
64560| {
64561|     NTSTATUS status = STATUS_SUCCESS;
64562|     RevertDebug(("UpdateRevertRecoveryInfo:
    | ssSeq=%08x, volDevObj=%08x, granule=%016l64x\n",
64563|     SnapShotSequenceNumber,
64564|     Volume,
64565|     LastGranuleFinished.QuadPart));
64566|
64567|     tRevertInfo revertInfo = {0};
64568|     revertInfo.SnapShotSequenceNumber =
    | SnapShotSequenceNumber;
64569|     revertInfo.LastKnownGranuleFinished =
    | LastGranuleFinished;
64570|     revertInfo.RevertFlags =

```



```

    | PSM_REVERT_FLAG_IN_PROGRESS;
64571|
64572| // NOTE: We did not set
    | PSM_REVERT_FLAG_CREATE_UNDO_SNAPSHOT
64573| //    on purpose: by the time we actually save
    | the
64574| //    recovery info to the header we either (a)
    | did want
64575| //    an undo snapshot and already created it
    | (and it will
64576| //    still be there when we boot), or (b) we
    | didn't want it.
64577|
64578| PersistentDictionary::SetRevertBootInfo (Volume,
    | revertInfo);
64579|
64580| status = PersistentDictionary::SaveHeader (Volume);
64581| RevertDebug(("UpdateRevertRecoveryInfo returning
    | %08x\n",status));
64582| ASSERT(NT_SUCCESS(status));
64583|
64584| return status;
64585| }
64586|
64587| //-----
    | -----
    | -
64588|
64589| NTSTATUS CancelRevertAtBootForVolume ( VolumeEntry
    | *volumeEntry, PDEVICE_OBJECT Volume )
64590| {
64591|     RevertDebug(("Entering CancelRevertAtBootForVolume;
    | volumeEntry=%08x, Volume=%08x\n",volumeEntry,Volume));
64592|     LARGE_INTEGER NullTime = {0};
64593|     bool isSystemVolume = volumeEntry ?
    | (volumeEntry->isSystemVolume==1) : false;
64594|
64595|     // The CancelRevertFlags allows us to determine
    | whether we want to finalize revert
64596|     // on the next boot (due to encountering system
    | drive).
64597|     // Finalize means log event and write to registry
    | that revert is complete.
64598|
64599|     ULONG CancelRevertFlags = isSystemVolume ?
    | PSM_REVERT_FLAG_FINALIZE_REVERT : 0;
64600|     RevertDebug(("CancelRevertAtBootForVolume
    | (SysRevert): CancelRevertFlags =
    | %08x\n",CancelRevertFlags));
64601|     NTSTATUS status = RevertVolumeAtNextBoot (Volume,

```

```

    | ULONG(0), NullTime, CancelRevertFlags);
64602|   if ( status == PSM_ERROR_REBOOT_NEEDED ) {
64603|       status = STATUS_SUCCESS;
64604|       // Do not risk I/O to system volume if it we
    | just reverted it.
64605|       if ( isSystemVolume ) {
64606|           RevertDebug(("CancelRevertAtBootForVolume
    | (SysRevert): deferring events and status change.\n"));
64607|       } else {
64608|           LogRevertEvent (
    | PSM_CANCELED_REVERT_AT_BOOT, status, NULL, 0 );
64609|           UpdateRevertStatus
    | (PSM_STATUS_REVERT_AT_BOOT);
64610|       }
64611|   }
64612|
64613|   RevertDebug(("CancelRevertAtBootForVolume returning
    | %08x\n",status));
64614|   return status;
64615| }
64616|
64617| //-----
    | -----
    | -
64618|
64619| NTSTATUS CancelRevertAtBootForAllVolumes (
64620|   pkSnapShotMaster  master )
64621| {
64622|   NTSTATUS status = STATUS_SUCCESS;
64623|   RevertDebug(("CancelRevertAtBootForAllVolumes:
    | master=%08x\n",master));
64624|
64625|   // Cancel revert at boot for every volume
    | referenced by 'master' independently.
64626|
64627|   PersistentDictionary::BeginUpdate();
64628|   __try {
64629|       GetSnapShotForRead();
64630|   __try {
64631|       pkSnapShotEntry p =
    | GetTopSnapShotForMaster ( &master->SnapShots );
64632|       while ( p ) {
64633|           status = CancelRevertAtBootForVolume (
    | NULL, p->DeviceObject );
64634|           if ( !NT_SUCCESS(status) ) {
64635|               RevertDebug(("CancelRevertAtBootForAllVolumes:
    | CancelRevertAtBootForVolume(%08x) returned
    | %08x\n",p,status));
64636|               DoneWithSnapShot(p);

```

```

64637|         break;
64638|     }
64639|
64640|     p = GetNextSnapShotForMaster (
        | &master->SnapShots, p );
64641|     }
64642| } __finally {
64643|     ReleaseSnapShotForRead();
64644| }
64645| } __finally {
64646|     PersistentDictionary::EndUpdate();
64647| }
64648|
64649| RevertDebug(("CancelRevertAtBootForAllVolumes: in
        | master=%08x, out status=%08x\n",master,status));
64650| return status;
64651| }
64652|
64653| //-----
        | -----
        | -
64654|
64655| NTSTATUS RevertAtNextBoot ( ULONG snapShotSequence,
        | LARGE_INTEGER snapShotTime )
64656| {
64657|     NTSTATUS status = STATUS_NOT_FOUND;
64658|     ULONG numScheduled = 0;
64659|     // Schedule revert at boot for every volume
64660|     // referenced by 'master' independently.
64661|
64662|     pkSnapShotMaster *masterList = 0;
64663|     ULONG numMasters = 0;
64664|     status = CreateMasterListForSequence (
        | snapShotSequence, snapShotTime, masterList, numMasters
        | );
64665|     if ( NT_SUCCESS(status) ) {
64666|         PersistentDictionary::BeginUpdate();
64667|         __try {
64668|             GetSnapShotForRead();
64669|             __try {
64670|                 for ( ULONG masterIndex=0; masterIndex
                    | < numMasters; ++masterIndex ) {
64671|                     pkSnapShotEntry p =
                        | GetTopSnapShotForMaster (
                            | &(masterList[masterIndex]->SnapShots) );
64672|                     while ( p ) {
64673|                         NTSTATUS scheduleStatus =
                            | RevertVolumeAtNextBoot ( p->DeviceObject,
                                | snapShotSequence, snapShotTime, 0 );
64674|                         if ( scheduleStatus ==

```

```

    | PSM_ERROR_REBOOT_NEEDED ) {
64675|         ++numScheduled;
64676|     } else if ( scheduleStatus !=
    | STATUS_SUCCESS ) {
64677|         status = scheduleStatus;
64678|     }
64679|     p = GetNextSnapShotForMaster (
    | &(masterList[masterIndex]->SnapShots), p );
64680|     }
64681|     }
64682|     } __finally {
64683|         ReleaseSnapShotForRead();
64684|     }
64685|     } __finally {
64686|         PersistentDictionary::EndUpdate();
64687|     }
64688|
64689|     FREE_POINTER(masterList);
64690|     masterList = 0;
64691|     numMasters = 0;
64692| }
64693|
64694| if ( NT_SUCCESS(status) ) {
64695|     if ( numScheduled > 0 ) {
64696|         status = PSM_ERROR_REBOOT_NEEDED;
64697|     }
64698| }
64699|
64700| RevertDebug(("RevertAtNextBoot: in sequence=%08x,
    | out status=%08x\n",snapShotSequence,status));
64701| return status;
64702| }
64703|
64704| //-----
    | -----
    | -
64705|
64706| NTSTATUS AnalyzeRevertCacheRequirements (
64707|     VolumeEntry    *volumeList,
64708|     ULONG          numVolumes )
64709| {
64710|     NTSTATUS status = STATUS_SUCCESS;
64711|     const ULONG SafetyMarginInBytes = 4096 * 1024;
64712|     const ULONG SafetyMarginInGranules =
        | SafetyMarginInBytes / GRANULE_SIZE;
64713|
64714|     // Find out if there is enough cache space right
        | now
64715|     // to perform the revert.
64716|

```

```

64717|   for ( ULONG volumeIndex=0; volumeIndex <
        | numVolumes; ++volumeIndex ) {
64718|       // Passing preTestFlag=true to
        | PerformRevertOperation makes it just
64719|       // figure out the upper limit on cache usage
        | without doing anything to the disk...
64720|
64721|       __try {
64722|           status = PerformRevertOperation (
        | volumeList[volumeIndex], 0, 0, true );
64723|       } __except(
        | ExceptionFilter(GetExceptionInformation()) ) {
64724|           status = GetExceptionCode();
64725|           RevertDebug(("!!!
        | AnalyzeRevertCacheRequirements: Exception %08x
        | occurred in pre-revert cache test\n",status));
64726|       }
64727|
64728|       if ( NT_SUCCESS(status) ) {
64729|           status = PSM_INSUFFICIENT_CACHE; //
        | assume worst case until we know it's safe
64730|           // The number of granules that would be
        | modified on disk is returned in
64731|           // volumeList[volumeIndex].cacheSlotsUsed.
64732|
        | RevertDebug(("AnalyzeRevertCacheRequirements:
        | volumeList[%08x].cacheSlotsUsed = %016l64x\n",
64733|           volumeIndex,
64734|
        | volumeList[volumeIndex].cacheSlotsUsed.QuadPart));
64735|
64736|           ASSERT (
        | volumeList[volumeIndex].cacheSlotsUsed.HighPart == 0 );
64737|           if (
        | volumeList[volumeIndex].cacheSlotsUsed.HighPart == 0 )
        | {
64738|               // Everything you ever wanted to know
        | about cache usage is in the device extension...
64739|               PFILTERED_EXTENSION devExt =
        | GetFilteredExtension(
        | volumeList[volumeIndex].volumeObject );
64740|
64741|               // Figure out how many cache slots are
        | available on the given volume.
64742|               __int64 CacheGranulesInUse =
        | __int64(devExt->Cache.CurrentCacheFileSize);
64743|               __int64 CacheGranulesTotal =
        | __int64(devExt->Cache.PSManBitMapSize);
64744|               __int64 CacheGranulesNeeded =
        | __int64(volumeList[volumeIndex].cacheSlotsUsed.QuadPart)

```

```

| ;
64745|
64746|     | RevertDebug(("AnalyzeRevertCacheRequirements:
| DevExt=%08x, CacheGranulesInUse=%08l64x,
| CacheGranulesTotal=%08l64x\n",devExt,CacheGranulesInUse,
| CacheGranulesTotal));
64747|         ASSERT ( CacheGranulesInUse <=
| CacheGranulesTotal );
64748|         if ( CacheGranulesInUse +
| CacheGranulesNeeded + SafetyMarginInGranules <
| CacheGranulesTotal ) {
64749|             status = STATUS_SUCCESS;
64750|
| RevertDebug(("AnalyzeRevertCacheRequirements:
| volumeList[%08x] passes cache test!\n",volumeIndex));
64751|         } else {
64752|
| RevertDebug(("AnalyzeRevertCacheRequirements: !!!
| volumeList[%08x] FAILS CACHE TEST !!!\n",volumeIndex));
64753|
| LogRevertEvent(PSM_INSUFFICIENT_CACHE_FOR_REVERT,PSM_INS
| UFFICIENT_CACHE_FOR_REVERT,NULL,0);
64754|         }
64755|     }
64756| }
64757|
64758|     if ( !NT_SUCCESS(status) ) {
64759|         break;
64760|     }
64761| }
64762|
64763| RevertDebug(("AnalyzeRevertCacheRequirements
| returning %08x\n",status));
64764| return status;
64765| }
64766|
64767| //-----
| -----
| -
64768|
64769| NTSTATUS RelocateSnapshotEntry (
64770|     pkSnapshotMaster  master,
64771|     VolumeEntry       &volumeEntry )
64772| {
64773|     NTSTATUS status = STATUS_NOT_FOUND;
64774|     volumeEntry.snapshot = NULL;
64775|     pkSnapshotEntry p = GetTopSnapshotForMaster (
| &master->Snapshots );
64776|     while ( p ) {

```

```

64777|     PFILTERED_EXTENSION devExt =
        | GetFilteredExtension(p->DeviceObject);
64778|     if ( devExt->Volumeld == volumeEntry.volumeld )
        | {
64779|         volumeEntry.snapshot = p;
64780|         status = STATUS_SUCCESS;
64781|         // NOTE: We deliberately refrain from
        | calling DoneWithSnapShot
64782|         //     so that we still have a reference
        | to the snapshot.
64783|         //     We will call
        | DoneWithRevertVolumes later.
64784|         volumeEntry.acquired = 1; // remember to
        | call DoneWithSnapShot later
64785|         break;
64786|     }
64787|
64788|     p =
        | GetNextSnapShotForMaster(&master->SnapShots,p);
64789| }
64790|
64791| return status;
64792| }
64793|
64794| //-----
        | -----
        | -
64795|
64796| NTSTATUS RegenerateVolumeListFromMasterList (
64797|     VolumeEntry      *volumeList,
64798|     ULONG             numVolumes,
64799|     pkSnapShotMaster  masterList[],
64800|     ULONG             numMasters )
64801| {
64802|     NTSTATUS status = STATUS_SUCCESS;
64803|     ASSERT ( masterList != NULL );
64804|     ASSERT ( numMasters > 0 );
64805|
64806|     GetSnapShotForRead();
64807|     __try {
64808|         for ( ULONG i=0; i < numVolumes; ++i ) {
64809|             status = STATUS_NOT_FOUND;
64810|             for ( ULONG m=0; m < numMasters; ++m ) {
64811|                 status = RelocateSnapShotEntry (
                    | masterList[m], volumeList[i] );
64812|                 if ( status == STATUS_SUCCESS ) {
64813|                     break;
64814|                 }
64815|             }
64816|

```

```

64817|         ASSERT ( status == STATUS_SUCCESS );
64818|         if ( status != STATUS_SUCCESS ) {
64819|             RevertDebug(("RegenerateVolumeList:
| error finding volumeId=%08x, index=%08x\n",
| volumeList[i].volumeId, i));
64820|             break;
64821|         }
64822|     }
64823| } __finally {
64824|     ReleaseSnapShotForRead();
64825| }
64826|
64827| RevertDebug(("RegenerateVolumeList returning
| %08x\n",status));
64828| return status;
64829| }
64830|
64831| //-----
| -----
| -
64832|
64833| NTSTATUS CreateMasterListForSequence (
64834|     ULONG          masterSequence,
64835|     LARGE_INTEGER  masterTime,
64836|     pkSnapShotMaster * &masterList,
64837|     ULONG          &numMastersInList )
64838| {
64839|     NTSTATUS status = STATUS_SUCCESS;
64840|     ULONG masterListArraySize = 1024;
64841|     do {
64842|         masterList = (pkSnapShotMaster *)
| MemAllocatePoolWithTag (
64843|             PagedPool,
64844|             sizeof(pkSnapShotMaster)*masterListArraySize,
64845|             TEMPTAG );
64846|
64847|         if ( masterList ) {
64848|             status = GetMasterListForSequence (
64849|                 masterSequence,
64850|                 masterTime,
64851|                 masterList,
64852|                 masterListArraySize,
64853|                 numMastersInList );
64854|
64855|             if ( status ==
| PSM_ERROR_INSUFFICIENT_BUFFER ) {
64856|                 // try an array with twice as many
| elements...
64857|                 MemFreePool(masterList);

```



```

64858|         masterList = 0;
64859|         numMastersInList = 0;
64860|         masterListArraySize *= 2;
64861|     }
64862| } else {
64863|     RevertDebug(("!!!
| CreateMasterListForSequence: Out of memory allocating
| master list!\n"));
64864|     status = STATUS_INSUFFICIENT_RESOURCES;
64865|     numMastersInList = masterListArraySize = 0;
64866|     break;
64867| }
64868| } while ( status == PSM_ERROR_INSUFFICIENT_BUFFER
| );
64869|
64870| #ifdef _DEBUG
64871|     RevertDebug(("CreateMasterListForSequence: in
| sequence=%08x time=%016l64x | out status=%08x,
| numMasters=%08x\n",masterSequence,masterTime.QuadPart,st
| atus,numMastersInList));
64872|     for ( ULONG masterIndex=0; masterIndex <
| numMastersInList; ++masterIndex ) {
64873|         RevertDebug((" masterList[%08x] =
| %08x\n",masterIndex,masterList[masterIndex]));
64874|     }
64875| #endif
64876|
64877|     if ( !NT_SUCCESS(status) ) {
64878|         // The caller should have to free memory only
| if we return success.
64879|         if ( masterList ) {
64880|             FREE_POINTER(masterList);
64881|             masterList = 0;
64882|         }
64883|
64884|         numMastersInList = 0;
64885|     }
64886|
64887|     return status;
64888| }
64889|
64890| //-----
| -----
| -
64891|
64892| NTSTATUS RegenerateVolumeList (
64893|     VolumeEntry *volumeList,
64894|     ULONG numVolumes,
64895|     ULONG masterSequence,
64896|     LARGE_INTEGER masterTime )

```

```

64897| {
64898|     NTSTATUS status = STATUS_SUCCESS;
64899|     ULONG numMastersInList = 0;
64900|     pkSnapshotMaster *masterList = 0;
64901|
64902|     status = CreateMasterListForSequence
        | (masterSequence, masterTime, masterList,
        | numMastersInList);
64903|     if ( NT_SUCCESS(status) ) {
64904|         ASSERT ( numMastersInList > 0 );
64905|         RegenerateVolumeListFromMasterList (
            | volumeList, numVolumes, masterList, numMastersInList );
64906|     } else {
64907|         numVolumes = 0;
64908|         RevertDebug(("!!!! Big problem %08x !!!!
            | Revert could not re-generate master list!\n",status));
64909|     }
64910|
64911|     if ( masterList ) {
64912|         MemFreePool (masterList);
64913|         masterList = 0;
64914|         numMastersInList = 0;
64915|     }
64916|
64917|     return status;
64918| }
64919|
64920| //-----
        | -----
        | -
64921|
64922| STATIC void Revert_CleanupOpenCloseResource (
64923|     ULONG        numVolumes,
64924|     VolumeEntry  *volumeList )
64925| {
64926|     if ( volumeList ) {
64927|         ASSERT (numVolumes > 0);
64928|         for ( ULONG volumeIndex=0; volumeIndex <
            | numVolumes; ++volumeIndex ) {
64929|             if ( volumeList[volumeIndex].openCloseDirty
                | ) {
64930|                 PFILTERED_EXTENSION devExt =
                    | GetFilteredExtension
                    | (volumeList[volumeIndex].volumeObject);
64931|
                    | RevertDebug(("Revert_ReleaseOpenCloseResource:
                    | index=%d, saved
                    | oca=%08x\n",volumeIndex,volumeList[volumeIndex].saveOpen
                    | CloseAcquired));
64932|                 devExt->OpenCloseAcquired =

```

```

        | volumeList[volumeIndex].saveOpenCloseAcquired;
64933|         volumeList[volumeIndex].openCloseDirty
        | = 0;
64934|     }
64935| }
64936| }
64937| }
64938|
64939| //-----
        | -----
        | -
64940|
64941| STATIC NTSTATUS SbRevertToSnapShot_Internal (
64942|     pkSnapShotMaster  master,
64943|     ULONG              revertFlags,
64944|     ULONG              &revertUndoSequence,
64945|     PDEVICE_OBJECT     volumeDeviceObject,
64946|     unsigned __int64    startingGranule )
64947| {
64948|     NTSTATUS status = STATUS_SUCCESS;
64949|     ULONG OpenCloseAcquired = FALSE;
64950|     ULONG numVolumes = 0;
64951|     VolumeEntry *volumeList = 0;
64952|
64953|     __try {
64954|         //!!!!!!!!!!!!!! turn off revert undo
        | snapshot for beta !!!!!!!!!!!!!!!
64955|         revertFlags &=
        | ~PSM_REVERT_FLAG_CREATE_UNDO_SNAPSHOT;
64956|         //!!!!!!!!!!!!!! turn off revert undo
        | snapshot for beta !!!!!!!!!!!!!!!
64957|
64958|         RevertDebug((
64959|             "Entering SbRevertToSnapShot_Internal;
        | master=%08x, volumeDevObj=%08x, flags=%08x\n",
64960|             master,
64961|             volumeDeviceObject,
64962|             revertFlags));
64963|
64964|         revertUndoSequence = 0;
64965|
64966|         //!!!!!!!!!!!!!!
        | !!!!!!!!!!!!!!!
64967|         // Hack to avoid multi-volume revert bug:
64968|         // If registry key EnableMultiVolumeRevert is
        | zero, do not allow revert
64969|         // on a snapshot with more than one volume in
        | it.
64970|         ULONG EnableMultiVolumeRevert = 0xf00f1e;

```

```

64971|
| Reg_GetULONGKey(&gRegistryPath,L"EnableMultiVolumeRevert
| ",0xf00f1e,&EnableMultiVolumeRevert);
64972|
| //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
| !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
64973|
64974|
| //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
| !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
64975| // Registry option for disabling revert
| on system drive:
64976| ULONG EnableSystemVolumeRevert = 0;
64977|
| Reg_GetULONGKey(&gRegistryPath,L"EnableSystemVolumeRever
| t",0,&EnableSystemVolumeRevert); // sys vol revert
| disabled by default
64978|
| //!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
| !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
64979|
64980| RevertDebug(("SbRevertToSnapShot_Internal:
| EnableMultiVolumeRevert=0x%x,
| EnableSystemVolumeRevert=0x%x\n",
64981| EnableMultiVolumeRevert,
64982| EnableSystemVolumeRevert));
64983|
64984| if ( Global_InRevert ) {
64985| status = STATUS_UNSUCCESSFUL;
64986| ASSERT ( !Global_InRevert );
64987| RevertDebug(("SbRevertToSnapShot_Internal:
| !!! Already in revert !!!\n"));
64988| } else {
64989| Global_InRevert = TRUE;
64990| __try {
64991| ULONG masterSequence = 0;
64992| status = GetSequenceForMaster (master,
| masterSequence);
64993| if ( !NT_SUCCESS(status) ) {
64994|
| RevertDebug(("SbRevertToSnapShot_Internal: Could not
| get sequence number for master!
| status=%08x\n",status));
64995| } else {
64996| LARGE_INTEGER masterTime =
| master->SnapShotTime;
64997|
| RevertDebug(("SbRevertToSnapShot_Internal: master=%08x
| has time=%016l64x\n",master,masterTime));
64998| if ( revertFlags &

```

```

    | PSM_REVERT_FLAG_ATBOOT ) {
64999|         status = RevertAtNextBoot
    | (masterSequence, masterTime);
65000|     } else if ( revertFlags &
    | PSM_REVERT_FLAG_CANCEL_ATBOOT ) {
65001|         status =
    | CancelRevertAtBootForAllVolumes (master);
65002|     } else {
65003|         const ULONG maxNumVolumes =
    | 512;
65004|         const ULONG
    | volumeListBytesAllocated = maxNumVolumes *
    | sizeof(VolumeEntry);
65005|         volumeList = (VolumeEntry *)
    | MemAllocatePoolWithTag (
65006|             PagedPool,
65007|             volumeListBytesAllocated,
65008|             TEMPTAG );
65009|
65010|         if ( !volumeList ) {
65011|
    | RevertDebug(("SbRevertToSnapShot_Internal: Out of
    | memory (volumeList)\n"));
65012|         status =
    | STATUS_INSUFFICIENT_RESOURCES;
65013|     } else {
65014|         __try {
65015|             // Make sure everything
    | in the array is initialized to zero (especially flags
    | and pointers).
65016|             RtlZeroMemory (
    | volumeList, volumeListBytesAllocated );
65017|
65018|             status =
    | PrepareVolumeList (
65019|                 master,
65020|                 volumeDeviceObject,
65021|                 volumeList,
65022|                 maxNumVolumes,
65023|                 numVolumes,
65024|
    | EnableSystemVolumeRevert );
65025|
65026|             if ( NT_SUCCESS(status)
    | ) {
65027|                 if (
    | !EnableMultiVolumeRevert && numVolumes>1 ) {
65028|
    | //////////////////////////////////////
    | //////////////////////////////////////

```

```

65029|                                     // hack to
    | avoid multi-volume revert issue
65030|
    | RevertDebug(("SbRevertToSnapShot_Internal:
    | Multi-volume revert disabled
    | (numVolumes=%08x)\n",numVolumes));
65031|                                     status =
    | PSM_MULTI_VOLUME_REVERT_DISABLED;
65032|                                     LogRevertEvent
    | ( PSM_MULTI_VOLUME_REVERT_DISABLED,
    | PSM_MULTI_VOLUME_REVERT_DISABLED, NULL, 0 );
65033|
    | UpdateLastRevertResult
    | (PSM_MULTI_VOLUME_REVERT_DISABLED);
65034|
    | //////////////////////////////////////
    | //////////////////////////////////////
65035|                                     } else {
65036|                                     if (
    | revertFlags & PSM_REVERT_FLAG_CREATE_UNDO_SNAPSHOT ) {
65037|                                     status =
    | CreateRevertUndoSnapShot (
65038|
    | volumeList,
65039|
    | numVolumes,
65040|
    | revertFlags,
65041|
    | revertUndoSequence );
65042|                                     }
65043|
65044|                                     if (
    | NT_SUCCESS(status) ) {
65045|                                     __try {
65046|                                     status
    | = AnalyzeRevertCacheRequirements (volumeList,
    | numVolumes);
65047|                                     } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
65048|                                     status
    | = GetExceptionCode();
65049|
    | RevertDebug(("!!! SbRevertToSnapShot_Internal:
    | Exception %08x in AnalyzeCacheRequirements\n",status));
65050|                                     }
65051|
65052|                                     if (
    | NT_SUCCESS(status) ) {
65053|                                     if (

```

```

    | revertFlags & PSM_REVERT_FLAG_DISMOUNT_AFFECTED_VOLUMES
    | ) {
65054|                                     //
65055|                                     //
    | !!!! WARNING WARNING WARNING !!!!
65056|                                     //
65057|                                     //
    | As soon as we do the dismount, we cannot use the
    | 'master'
65058|                                     //
    | pointer anymore. Must use masterSequence to refer to
65059|                                     //
    | the master in question.
65060|                                     //
65061|
    | status = DismountVolumeList (volumeList, numVolumes);
65062|                                     }
65063|
65064|                                     master
    | = 0; //make sure we don't use pointer (see warning
    | above)
65065|
65066|                                     if (
    | NT_SUCCESS(status) ) {
65067|
    | NTSTATUS AcquireStatus = STATUS_WAIT_0;
65068|                                     if
    | ( revertFlags &
    | PSM_REVERT_FLAG_DISMOUNT_AFFECTED_VOLUMES ) {
65069|
    | // If we were told to dismount the volumes, it means
    | that we are not doing this at boot.
65070|
    | // Therefore we need to acquire the OpenClose resource.
65071|
    | // We must *NOT* acquire OpenClose when booting up
    | because LoadSnapShotsForVolume
65072|
    | // has already done this.
65073|
65074|
    | AcquireStatus = AcquireOpenCloseResource();
65075|
    | if ( AcquireStatus == STATUS_WAIT_0 ) {
65076|
    | OpenCloseAcquired = TRUE;
65077|
    | RevertDebug(("SbRevertToSnapshot_Internal: OpenClose
    | resource has been acquired.\n"));
65078|

```

```

    | } else {
65079|
    | RevertDebug(("SbRevertToSnapShot_Internal: !!!
    | OpenClose resource NOT acquired:
    | %08x\n",AcquireStatus));
65080|
    | }
65081|                                     }
65082|
65083|                                     if
    | ( AcquireStatus == STATUS_WAIT_0 ) {
65084|
    | for ( ULONG volumeIndex=0; volumeIndex < numVolumes;
    | ++volumeIndex ) {
65085|
    | // pkSnapShotEntry p =
    | volumeList[volumeIndex].snapshot;
65086|
    | PFILTERED_EXTENSION devExt = GetFilteredExtension
    | (volumeList[volumeIndex].volumeObject);
65087|
    | if ( OpenCloseAcquired ) {
65088|
    | volumeList[volumeIndex].saveOpenCloseAcquired =
    | devExt->OpenCloseAcquired;
65089|
    | volumeList[volumeIndex].openCloseDirty = 1;
65090|
    | devExt->OpenCloseAcquired = TRUE;
65091|
    | }
65092|
    | RevertDebug(("SbRevertToSnapShot_Internal: About to
    | call PerformRevertOperation with
    | volumeIndex=%08x\n",volumeIndex));
65093|
65094|
    | __try {
65095|
    | status = PerformRevertOperation (
65096|
    | volumeList[volumeIndex],
65097|
    | revertFlags,
65098|
    | startingGranule,
65099|
    | false );
65100|
    | } __except( ExceptionFilter(GetExceptionInformation())

```



```

    | ) {
65101|
    | status = GetExceptionCode();
65102|
    | RevertDebug(("!!! SbRevertToSnapShot_Internal:
    | Exception %08x in PerformRevertOperation\n",status));
65103|
    | }
65104|
65105|
    | startingGranule = 0;
65106|
    | if ( !NT_SUCCESS(status) ) {
65107|
    | RevertDebug(("!!! SbRevertToSnapShot_Internal: breaking
    | out of volume loop early due to error %08x\n",status));
65108|
    | break;
65109|
    | }
65110|
    | }
65111|
    | }
    | else {
65112|
    | RevertDebug(("!!! SbRevertToSnapShot_Internal: Could
    | not acquire Open/Close resource
    | (%08x)\n",AcquireStatus));
65113|
    | status = PSM_CANCELED_BY_USER; //??? Is this really
    | the right thing to return ???
65114|
    | }
65115|
    | } else
    | {
65116|
    | status = RevertAtNextBoot (masterSequence, masterTime);
65117|
    | }
65118|
    | } else {
65119|
    | // Not
    | enough cache to perform the revert...
65120|
    | if (
    | revertFlags & PSM_REVERT_FLAG_CREATE_UNDO_SNAPSHOT ) {
65121|
    | //
    | !!! Delete the revert undo snapshot
65122|
    | ASSERT(FALSE);
65123|
    | }
65124|
    | }
65125|
    | }
65126|
    | }

```

```

65127|         }
65128|
65129|         if ( revertFlags &
        | PSM_REVERT_FLAG_DISMOUNT_AFFECTED_VOLUMES ) {
65130|             // We are doing a
        | command line revert, not a revert at boot.
65131|             // Before
        | remounting the dismounted volumes, we must let go of
        | their snapshots.
65132|             DoneWithRevertVolumes ( volumeList, numVolumes, false
        | );
65133|
65134|             // Must remount
        | whether or not an error occurred...
65135|             // The call will
        | only remount volumes whose VolumeEntry::dismounted
65136|             // flag is set,
        | meaning they really were dismounted.
65137|             RemountAffectedVolumes ( volumeList, numVolumes,
        | revertFlags );
65138|
65139|             if (
        | OpenCloseAcquired ) {
65140|                 __try {
65141|                     Revert_CleanupOpenCloseResource (numVolumes,
        | volumeList);
65142|                 } __finally {
65143|                     ReleaseOpenCloseResource();
65144|                     OpenCloseAcquired = FALSE;
65145|                     RevertDebug(("SbRevertToSnapShot_Internal: post-revert
        | (1): OpenClose resource has been released.\n"));
65146|                 }
65147|             }
65148|
65149|             // *** IMPORTANT
        | *** The remount has caused all of our
65150|             // snapshot
        | pointers and master pointers to become invalid.
65151|             // We need to
        | re-acquire everything based on persistent identifiers
65152|             // like snapshot
        | sequence numbers and volume IDs.
65153|             NTSTATUS
        | reGenStatus = RegenerateVolumeList ( volumeList,

```

```

    | numVolumes, masterSequence, masterTime );
65154|         if (
    | !NT_SUCCESS(reGenStatus) ) {
65155|             numVolumes = 0;
    | // keep us from using invalid volume entries
65156|         }
65157|     }
65158| } __finally {
65159|     DoneWithRevertVolumes (
    | volumeList, numVolumes, true );
65160| }
65161| }
65162| }
65163| }
65164| } __finally {
65165|     __try {
65166|         if ( OpenCloseAcquired ) {
65167|             __try {
65168|
    | Revert_CleanupOpenCloseResource (numVolumes,
    | volumeList);
65169|         } __finally {
65170|             ReleaseOpenCloseResource();
65171|             OpenCloseAcquired = FALSE;
65172|
    | RevertDebug(("SbRevertToSnapShot_Internal: post-revert
    | (2): OpenClose resource has been released.\n"));
65173|         }
65174|     }
65175| } __finally {
65176|     Global_InRevert = FALSE;
65177|     if ( volumeList ) {
65178|         FREE_POINTER (volumeList);
65179|
    | RevertDebug(("SbRevertToSnapShot_Internal: volumeList
    | array has been freed.\n"));
65180|     }
65181|     numVolumes = 0;
65182| }
65183| }
65184| }
65185| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
65186|     status=GetExceptionCode();
65187|     RevertDebug(("SbRevertToSnapShot_Internal:
    | Error! Exception %08x\n",status));
65188| }
65189|
65190|     RevertDebug(("SbRevertToSnapShot_Internal returning
    | 0x%08x\n",status));

```

```

65191|    return status;
65192| }
65193|
65194| //-----
| -----
| -
65195|
65196| void SbRevertThreadFunc ( RevertThreadParms
| *revertParms )
65197| {
65198|    __try {
65199|        if ( revertParms != NULL ) {
65200|            Debug(DEBUG_DEVCON,(
65201|                "Entering SbRevertThreadFunc;
| revertParms=%08x, KernelPointer=%08x\n",
65202|                revertParms,
65203|                revertParms->Master));
65204|
65205|            revertParms->Status =
| SbRevertToSnapShot_Internal (
65206|                revertParms->Master,
65207|                revertParms->Flags,
65208|                revertParms->RevertUndoSequence,
65209|                revertParms->VolumeDeviceObject,
65210|                revertParms->StartingGranule.QuadPart
| );
65211|        } else {
65212|            RevertDebug(("!!! SbRevertThreadFunc:
| revertParms==NULL !!!\n"));
65213|        }
65214|    } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
65215|        // Not much we can do here... the exception
| might have been caused
65216|        // by revertParms being an invalid pointer, so
| don't try to set return code.
65217|        NTSTATUS code = GetExceptionCode();
65218|        RevertDebug(("!!! Exception %08x in
| SbRevertThreadFunc !!!\n",code));
65219|    }
65220| }
65221|
65222| //-----
| -----
| -
65223|
65224| NTSTATUS SbRevertToSnapShot_SeparateThread_Internal (
65225|    tkSnapShotMaster *master,
65226|    PDEVICE_OBJECT volumeDeviceObject,
65227|    ULONG revertFlags,

```

```

65228|    ULONG          &revertUndoSequence,
65229|    unsigned __int64  startingGranule )
65230| {
65231|    NTSTATUS Status = STATUS_PENDING;
65232|    RevertThreadParms RevertParms = {0};
65233|    RevertParms.Master          = master;
65234|    RevertParms.Flags           =
        | revertFlags;
65235|    RevertParms.Status          =
        | STATUS_PENDING;
65236|    RevertParms.RevertUndoSequence = 0;
65237|    RevertParms.StartingGranule.QuadPart =
        | startingGranule;
65238|    RevertParms.VolumeDeviceObject =
        | volumeDeviceObject;
65239|
65240|    Debug(DEBUG_DEVCON,("PSMAN: About to start revert
        | thread...\n"));
65241|    HANDLE RevertThreadHandle = NULL;
65242|    Status = pmStartThread(
65243|        (PKSTART_ROUTINE)SbRevertThreadFunc,
65244|        &RevertParms,
65245|        &RevertThreadHandle );
65246|
65247|    if(NT_SUCCESS(Status)) {
65248|        Debug(DEBUG_DEVCON,("PSMAN: Revert thread has
        | been started.\n"));
65249|
        | ZwWaitForSingleObject(RevertThreadHandle,FALSE,NULL);
65250|        ZwClose(RevertThreadHandle);
65251|        RevertThreadHandle = NULL;
65252|        Status = RevertParms.Status;
65253|        Debug(DEBUG_DEVCON,("PSMAN: Revert returned
        | status=%08x\n",Status));
65254|    } else {
65255|        Debug(DEBUG_DEVCON,("Error %08x starting revert
        | thread\n",Status));
65256|    }
65257|
65258|    revertUndoSequence =
        | RevertParms.RevertUndoSequence;
65259|    return Status;
65260| }
65261|
65262| //-----
        | -----
        | -
65263|
65264| NTSTATUS SbRevertToSnapShot_SeparateThread (
65265|    tkSnapShotMaster *master,

```

```

65266| PDEVICE_OBJECT    volumeDeviceObject,
65267| ULONG              revertFlags,
65268| ULONG              &revertUndoSequence )
65269| {
65270|     NTSTATUS status = STATUS_SUCCESS;
65271|
65272|     __try {
65273|         status =
        | SbRevertToSnapShot_SeparateThread_Internal (
65274|             master,
65275|             volumeDeviceObject,
65276|             revertFlags,
65277|             revertUndoSequence,
65278|             0 );
65279|     } __except(
        | ExceptionFilter(GetExceptionInformation()) ) {
65280|         status = GetExceptionCode();
65281|         RevertDebug(("!!!
        | SbRevertToSnapShot_SeparateThread: Exception %08x
        | !!!\n",status));
65282|     }
65283|
65284|     return status;
65285| }
65286|
65287| //-----
        | -----
        | -
65288|
65289| void MyDumpRevertInfo ( pRevertInfo r )
65290| {
65291|     RevertDebug(("  RevertInfo = %08x :\n", r ));
65292|     RevertDebug(("    SnapSeqNum = %08x\n",
        | r->SnapShotSequenceNumber));
65293|     RevertDebug(("    LastGran  = %016l64x\n",
        | r->LastKnownGranuleFinished));
65294|     RevertDebug(("    Flags    = %08x\n",
        | r->RevertFlags));
65295|     RevertDebug(("    SnapTime  = %016l64x\n",
        | r->SnapShotTime));
65296| }
65297|
65298| //-----
        | -----
        | -
65299|
65300| void MyDumpHeader ( pHeader header )
65301| {
65302|     RevertDebug(("Dump of header=%08x :\n", header));
65303|     RevertDebug(("  Version  = %08x\n",

```

```

    | header->Version));
65304|   RevertDebug(("   Size      = %08x\n",
    | header->Size));
65305|   RevertDebug(("   Signature = %08x\n",
    | header->Signature));
65306|   RevertDebug(("   GranSize  = %08x\n",
    | header->GranuleSizeInBytes));
65307|   RevertDebug(("   LoadWheel = %08x\n",
    | header->IndexLoadWheel));
65308|   RevertDebug(("   HiSnapNum = %08x\n",
    | header->HighestSnapNumber));
65309|   RevertDebug(("   DateTimeWr = %016l64x\n",
    | header->DateTimeWritten));
65310|   MyDumpRevertInfo ( &(header->RevertInfo) );
65311| }
65312|
65313| //-----
    | -----
    | -
65314|
65315| STATIC ULONG IsSystemVolume ( const WCHAR *DeviceName )
65316| {
65317|   UNICODE_STRING SystemPartitionString = {0};
65318|   UNICODE_STRING RegistryPathString = {0};
65319|   const WCHAR * const RegistryPath =
    | L"\\Registry\\Machine\\System\\Setup\\";
65320|   RtlInitUnicodeString (&RegistryPathString,
    | RegistryPath);
65321|
    | Reg_GetStringKey(&RegistryPathString,L"SystemPartition",
    | L"",&SystemPartitionString);
65322|   RevertDebug(("IsSystemVolume: Found
    | SystemPartition='%S' in
    | registry\n",SystemPartitionString.Buffer));
65323|   ULONG system =
    | (wcscmp(SystemPartitionString.Buffer,DeviceName)==0) ?
    | TRUE : FALSE;
65324|   Reg_FreeString(&SystemPartitionString);
65325|   RevertDebug(("IsSystemVolume '%S' returning
    | %08x\n",DeviceName,system));
65326|   return system;
65327| }
65328|
65329| //-----
    | -----
    | -
65330|
65331| extern "C" {
65332|   NTKERNELAPI void InbvEnableDisplayString ( DWORD
    | Flag );

```

```

65333| }
65334|
65335| NTSTATUS RevertCheckAtVolumeMount ( PDEVICE_OBJECT
    | volumeDeviceObject )
65336| {
65337|     NTSTATUS status = STATUS_SUCCESS;
65338|     ULONG DoReboot=FALSE;
65339|     RevertDebug(("RevertCheckAtVolumeMount:
    | volumeDevObj=%08x\n", volumeDeviceObject));
65340|
65341|     ASSERT ( volumeDeviceObject != NULL );
65342|     PFILTERED_EXTENSION devExt = GetFilteredExtension
    | (volumeDeviceObject);
65343|     ASSERT ( devExt != NULL );
65344|     ULONG isSystemVolume =
    | IsSystemVolume(devExt->Name);
65345|
65346|     if ( Global_InRevert ) {
65347|         RevertDebug(("RevertCheckAtVolumeMount:
    | Already in revert; ignoring check.\n"));
65348|     } else {
65349|         ULONG revertUndoSequence = 0;
65350|
65351|         pHeader header = devExt->Cache.Header;
65352|         if ( header ) {
65353|             MyDumpHeader(header);
65354|             tRevertInfo &revertInfo =
    | header->RevertInfo;
65355|             if ( revertInfo.SnapShotSequenceNumber != 0
    | ) {
65356|                 // Looks like we have been requested to
    | do revert for this volume...
65357|                 RevertDebug(("RevertCheckAtVolumeMount:
    | !!! Found revert request !!!  seq=%08x,
    | time=%016l64x\n",
65358|                     revertInfo.SnapShotSequenceNumber,
65359|                     revertInfo.SnapShotTime.QuadPart));
65360|
65361|                 Global_NeedResultUpdate = TRUE;
65362|
65363|                 pkSnapShotMaster masterToRevertTo = 0;
65364|
65365|                 status = GetMasterForSequenceAndVolume
    | (
65366|                     revertInfo.SnapShotSequenceNumber,
65367|                     revertInfo.SnapShotTime,
65368|                     volumeDeviceObject,
65369|                     masterToRevertTo );
65370|
65371|                 if ( NT_SUCCESS(status) ) {

```



```

65372|          ASSERT ( masterToRevertTo != NULL
| );
65373|          if ( masterToRevertTo ) {
65374|              status =
| SbRevertToSnapShot_SeparateThread_Internal (
65375|                  masterToRevertTo,
65376|                  volumeDeviceObject,
65377|                  revertInfo.RevertFlags,
65378|                  revertUndoSequence,
65379|                  | revertInfo.LastKnownGranuleFinished.QuadPart );
65380|                  if(NT_SUCCESS(status)) {
65381|                      DoReboot = TRUE;
65382|                  }
65383|              } else {
65384|                  RevertDebug(("!!!
| RevertCheckAtVolumeMount: internal error: status=%08x
| but masterToRevertTo=NULL\n",status));
65385|                  status = STATUS_NOT_FOUND;
65386|              }
65387|          } else {
65388|              status = STATUS_NOT_FOUND;
65389|
| RevertDebug(("RevertCheckAtVolumeMount: could not find
| master for sequence number %08x\n",
65390|              | revertInfo.SnapShotSequenceNumber));
65391|          }
65392|      }
65393|  } else {
65394|      RevertDebug(("RevertCheckAtVolumeMount:
| header==NULL for
| volDevObj=%08x\n",volumeDeviceObject));
65395|      status = STATUS_INVALID_PARAMETER;
65396|  }
65397| }
65398|
65399| if((DoReboot) &&
| (!PersistentDictionary::GetSystemReady())) {
65400|     // The revert has been performed, and we are
| before system ready time.
65401|     // See if the volume we just reverted is the
| system volume.
65402|     if ( isSystemVolume ) {
65403|         Global_NeedResultUpdate = FALSE;
65404|
65405|         #if 0
65406|         | //-----
| -----

```

```

65407|          // Before generating intentional BSOD,
        | give system a delay to do hardware flush (?)
65408|
        | //-----
        | -----
65409|          const int SecondsToWait = 10;
65410|          LARGE_INTEGER TimeToWait;
65411|          TimeToWait.QuadPart =
        | RELATIVE(SECONDS(SecondsToWait));
65412|          KeDelayExecutionThread
        | ((KPROCESSOR_MODE)KernelMode, FALSE, &TimeToWait);
65413|
        | //-----
        | -----
65414|          #endif
65415|
65416|          #if 0
65417|
        | //-----
        | -----
65418|          // Cause an intentional blue screen of
        | death...
65419|          // We must in order to prevent booting
        | with both old drive data (before revert)
65420|          // mixed with post-revert data after
        | the revert.
65421|
        | //-----
        | -----
65422|          KeBugCheck(PSM_REVERT_COMPLETE);
65423|          #endif
65424|
65425|          #if 1
65426|
        | //-----
        | -----
65427|          // This code is for displaying a
        | human-readable message instead of causing an
        | intentional
65428|          // BSOD.
65429|
        | //-----
        | -----
65430|          InbvEnableDisplayString(1);
65431|          char message[] = "Please restart the
        | computer to boot with reverted system drive.\n";
65432|          HalDisplayString (message);
65433|          RevertDebug(("RevertCheckAtVolumeMount:
        | Just called HalDisplayString to notify user to
        | reboot\n"));

```

```

65434|         for(;;) {
65435|
65436|             | RevertDebug(("RevertCheckAtVolumeMount: SYSTEM IS HUNG
65437|             | - WAITING FOR POST-REVERT RESTART.\n"));
65438|             const int SecondsToWait = 30;
65439|             LARGE_INTEGER TimeToWait;
65440|             TimeToWait.QuadPart =
65441|             | RELATIVE(SECONDS(SecondsToWait));
65442|             KeDelayExecutionThread
65443|             | ((KPROCESSOR_MODE)KernelMode, FALSE, &TimeToWait);
65444|         }
65445|     #endif
65446| }
65447| }
65448|
65449| if ( Global_NeedResultUpdate ) {
65450|     UpdateLastRevertResult(status);
65451| }
65452|
65453| RevertDebug(("RevertCheckAtVolumeMount returning
65454|             | %08x\n",status));
65455| return status;
65456| }
65457|
65458| //-----
65459| | -----
65460| | -
65461|
65462| void LogRevertEvent (
65463|     ULONG     message,
65464|     NTSTATUS  error,
65465|     WCHAR     *strings[],
65466|     ULONG     numStrings )
65467| {
65468|     LogError (
65469|         (PDEVICE_OBJECT) PSMAN_DRIVER_OBJECT,
65470|         NULL,
65471|         message,
65472|         error,
65473|         NULL,
65474|         0,
65475|         strings,
65476|         numStrings );
65477|
65478|     RevertDebug(("LogRevertEvent: message=%08x,
65479|                 | error=%08x,
65480|                 | numStrings=%08x\n",message,error,numStrings));
65481|     for ( ULONG i=0; i<numStrings; ++i ) {
65482|         RevertDebug(("LogRevertEvent: string[%d] =
65483|                 | '%S'\n", i, strings[i]));

```

```

65474|    }
65475| }
65476|
65477| //-----
| -----
| -
65478|
65479| NTSTATUS UpdateRevertStatus ( DWORD Status )
65480| {
65481|     return UpdateDriverSubSystemStatus ( Status,
        | L"RevertStatus" );
65482| }
65483|
65484| //-----
| -----
| -
65485|
65486| NTSTATUS UpdateLastRevertResult ( DWORD ErrorCode )
65487| {
65488|     Global_NeedResultUpdate = FALSE;
65489|     if ( NT_SUCCESS(ErrorCode) ) {
65490|         ErrorCode = PSM_OPERATION_SUCCESSFUL;
65491|     }
65492|     return UpdateDriverSubSystemStatus ( ErrorCode,
        | L"LastRevertResult" );
65493| }
65494|
65495| //-----
| -----
| -
65496|
65497| /*--- end of file revert.cpp ---*/
65498|
65499|
65500|
65501| File Listing: revert.h
65502|
65503| //-----
| -----
65504| // SbRevertVolumeToSnapShot
65505| // Passed in a pkSnapShotMaster representing the
        | snapshot of volume(s)
65506| // to be reverted to. Only the specified volume is
        | reverted.
65507| //
65508| // NOTE: 'revertUndoSequence' is the where the
        | sequence number of the revert
65509| //     undo snapshot that was created by this call
        | will be put.
65510| //     If no undo snapshot was created, this value

```

```

    | will be set to zero.
65511| //-----
    | -----
65512| NTSTATUS SbRevertVolumeToSnapShot (
65513|     PDEVICE_OBJECT    volume,
65514|     pkSnapShotMaster   master,
65515|     ULONG               revertFlags,
65516|     ULONG               &revertUndoSequence );
65517|
65518|
65519| //-----
    | -----
65520| // SbRevertVolumeToSnapShot_SeparateThread
65521| // Creates a separate thread for
    | SbRevertVolumeToSnapShot to run in,
65522| // and waits for it to complete. This is so that Sb
    | functions
65523| // for doing I/O are allowed access to the cache file
    | handle.
65524| //
65525| // NOTE: 'revertUndoSequence' is the where the
    | sequence number of the revert
65526| //     undo snapshot that was created by this call
    | will be put.
65527| //     If no undo snapshot was created, this value
    | will be set to zero.
65528| //-----
    | -----
65529| NTSTATUS SbRevertVolumeToSnapShot_SeparateThread (
65530|     PDEVICE_OBJECT    volume,
65531|     pkSnapShotMaster   master,
65532|     ULONG               revertFlags,
65533|     ULONG               &revertUndoSequence );
65534|
65535| //-----
    | -----
65536| // SbRevertToSnapShot_SeparateThread
65537| // Creates a separate thread for SbRevertToSnapShot to
    | run in,
65538| // and waits for it to complete. This is so that Sb
    | functions
65539| // for doing I/O are allowed access to the cache file
    | handle.
65540| //
65541| // NOTE: 'revertUndoSequence' is the where the
    | sequence number of the revert
65542| //     undo snapshot that was created by this call
    | will be put.
65543| //     If no undo snapshot was created, this value
    | will be set to zero.

```

```

65544| //-----
    | -----
65545| NTSTATUS SbRevertToSnapShot_SeparateThread (
65546|     pkSnapShotMaster  master,
65547|     PDEVICE_OBJECT    volumeDeviceObject,
65548|     ULONG              revertFlags,
65549|     ULONG              &revertUndoSequence );
65550|
65551| //-----
    | -----
65552| // RevertCheckAtVolumeMount
65553| //
65554| // This function is called when PSM mounts a volume.
65555| // It determines whether a revert operation needs to be
65556| // initiated/completed on that volume, based on
    | information in
65557| // the header associated with that volume.
65558| //-----
    | -----
65559| NTSTATUS RevertCheckAtVolumeMount ( PDEVICE_OBJECT
    | volumeDeviceObject );
65560|
65561|
65562| //-----
    | -----
65563| // CancelRevertAtBootForAllVolumes
65564| //
65565| // This function cancels revert at boot for all volumes
    | that
65566| // have a revert scheduled.
65567| //-----
    | -----
65568| NTSTATUS CancelRevertAtBootForAllVolumes();
65569|
65570| //-----
    | -----
65571| // CancelRevertAtBootForVolume
65572| //
65573| // This function cancels revert at boot for a specified
    | volume.
65574| //-----
    | -----
65575| struct VolumeEntry;
65576| NTSTATUS CancelRevertAtBootForVolume ( VolumeEntry *,
    | PDEVICE_OBJECT );
65577|
65578| /*--- end of file revert.h ---*/
65579|
65580|
65581|

```

```

65582| File Listing: RTL.h
65583|
65584| #if _WIN32_WINNT<0x0500
65585| //
65586| // BitMap routines. The following structure,
    | routines, and macros are
65587| // for manipulating bitmaps. The user is responsible
    | for allocating a bitmap
65588| // structure (which is really a header) and a buffer
    | (which must be longword
65589| // aligned and multiple longwords in size).
65590| //
65591|
65592| typedef struct _RTL_BITMAP {
65593|     ULONG SizeOfBitMap;           // Number
    | of bits in bit map
65594|     PULONG Buffer;                // Pointer
    | to the bit map itself
65595| } RTL_BITMAP;
65596| typedef RTL_BITMAP *PRTL_BITMAP;
65597|
65598| //
65599| // The following routine initializes a new bitmap. It
    | does not alter the
65600| // data currently in the bitmap. This routine must be
    | called before
65601| // any other bitmap routine/macro.
65602| //
65603|
65604| NTSYSAPI
65605| VOID
65606| NTAPI
65607| RtlInitializeBitMap (
65608|     PRTL_BITMAP BitMapHeader,
65609|     PULONG BitMapBuffer,
65610|     ULONG SizeOfBitMap
65611| );
65612|
65613| //
65614| // The following two routines either clear or set all
    | of the bits
65615| // in a bitmap.
65616| //
65617|
65618| NTSYSAPI
65619| VOID
65620| NTAPI
65621| RtlClearAllBits (
65622|     PRTL_BITMAP BitMapHeader
65623| );

```

```

65624|
65625| NTSYSAPI
65626| VOID
65627| NTAPI
65628| RtlSetAllBits (
65629|     PRTL_BITMAP BitMapHeader
65630| );
65631|
65632| //
65633| // The following two routines locate a contiguous
        | region of either
65634| // clear or set bits within the bitmap. The region
        | will be at least
65635| // as large as the number specified, and the search of
        | the bitmap will
65636| // begin at the specified hint index (which is a bit
        | index within the
65637| // bitmap, zero based). The return value is the bit
        | index of the located
65638| // region (zero based) or -1 (i.e., 0xffffffff) if
        | such a region cannot
65639| // be located
65640| //
65641|
65642| NTSYSAPI
65643| ULONG
65644| NTAPI
65645| RtlFindClearBits (
65646|     PRTL_BITMAP BitMapHeader,
65647|     ULONG NumberToFind,
65648|     ULONG HintIndex
65649| );
65650|
65651| NTSYSAPI
65652| ULONG
65653| NTAPI
65654| RtlFindSetBits (
65655|     PRTL_BITMAP BitMapHeader,
65656|     ULONG NumberToFind,
65657|     ULONG HintIndex
65658| );
65659|
65660| //
65661| // The following two routines locate a contiguous
        | region of either
65662| // clear or set bits within the bitmap and either set
        | or clear the bits
65663| // within the located region. The region will be as
        | large as the number
65664| // specified, and the search for the region will begin

```



```

    | at the specified
65665| // hint index (which is a bit index within the bitmap,
    | zero based). The
65666| // return value is the bit index of the located region
    | (zero based) or
65667| // -1 (i.e., 0xffffffff) if such a region cannot be
    | located. If a region
65668| // cannot be located then the setting/clearing of the
    | bitmap is not performed.
65669| //
65670|
65671| NTSYSAPI
65672| ULONG
65673| NTAPI
65674| RtlFindClearBitsAndSet (
65675|     PRTL_BITMAP BitMapHeader,
65676|     ULONG NumberToFind,
65677|     ULONG HintIndex
65678| );
65679|
65680| NTSYSAPI
65681| ULONG
65682| NTAPI
65683| RtlFindSetBitsAndClear (
65684|     PRTL_BITMAP BitMapHeader,
65685|     ULONG NumberToFind,
65686|     ULONG HintIndex
65687| );
65688|
65689| //
65690| // The following two routines clear or set bits within
    | a specified region
65691| // of the bitmap. The starting index is zero based.
65692| //
65693|
65694| NTSYSAPI
65695| VOID
65696| NTAPI
65697| RtlClearBits (
65698|     PRTL_BITMAP BitMapHeader,
65699|     ULONG StartingIndex,
65700|     ULONG NumberToClear
65701| );
65702|
65703| NTSYSAPI
65704| VOID
65705| NTAPI
65706| RtlSetBits (
65707|     PRTL_BITMAP BitMapHeader,
65708|     ULONG StartingIndex,

```

```

65709|    ULONG NumberToSet
65710|    );
65711|
65712| //
65713| // The following two routines locate the longest
        | contiguous region of
65714| // clear or set bits within the bitmap. The returned
        | starting index value
65715| // denotes the first contiguous region located
        | satisfying our requirements
65716| // The return value is the length (in bits) of the
        | longest region found.
65717| //
65718|
65719| NTSYSAPI
65720| ULONG
65721| NTAPI
65722| RtlFindLongestRunClear (
65723|     PRTL_BITMAP BitMapHeader,
65724|     PULONG StartingIndex
65725| );
65726|
65727| NTSYSAPI
65728| ULONG
65729| NTAPI
65730| RtlFindLongestRunSet (
65731|     PRTL_BITMAP BitMapHeader,
65732|     PULONG StartingIndex
65733| );
65734|
65735| //
65736| // The following two routines locate the first
        | contiguous region of
65737| // clear or set bits within the bitmap. The returned
        | starting index value
65738| // denotes the first contiguous region located
        | satisfying our requirements
65739| // The return value is the length (in bits) of the
        | region found.
65740| //
65741|
65742| NTSYSAPI
65743| ULONG
65744| NTAPI
65745| RtlFindFirstRunClear (
65746|     PRTL_BITMAP BitMapHeader,
65747|     PULONG StartingIndex
65748| );
65749|
65750| NTSYSAPI

```

```

65751| ULONG
65752| NTAPI
65753| RtlFindFirstRunSet (
65754|     PRTL_BITMAP BitMapHeader,
65755|     PULONG StartingIndex
65756| );
65757|
65758| //
65759| // The following macro returns the value of the bit
        | stored within the
65760| // bitmap at the specified location. If the bit is
        | set a value of 1 is
65761| // returned otherwise a value of 0 is returned.
65762| //
65763| //     ULONG
65764| //     RtlCheckBit (
65765| //         PRTL_BITMAP BitMapHeader,
65766| //         ULONG BitPosition
65767| //     );
65768| //
65769| //
65770| // To implement CheckBit the macro retrieves the
        | longword containing the
65771| // bit in question, shifts the longword to get the bit
        | in question into the
65772| // low order bit position and masks out all other
        | bits.
65773| //
65774|
65775| #define RtlCheckBit(BMH,BP) (((BMH->Buffer[(BP) /
        | 32]) >> ((BP) % 32)) & 0x1)
65776|
65777| //
65778| // The following two procedures return to the caller
        | the total number of
65779| // clear or set bits within the specified bitmap.
65780| //
65781|
65782| NTSYSAPI
65783| ULONG
65784| NTAPI
65785| RtlNumberOfClearBits (
65786|     PRTL_BITMAP BitMapHeader
65787| );
65788|
65789| NTSYSAPI
65790| ULONG
65791| NTAPI
65792| RtlNumberOfSetBits (
65793|     PRTL_BITMAP BitMapHeader

```

```

65794|    );
65795|
65796| //
65797| // The following two procedures return to the caller a
        | boolean value
65798| // indicating if the specified range of bits are all
        | clear or set.
65799| //
65800|
65801| NTSYSAPI
65802| BOOLEAN
65803| NTAPI
65804| RtlAreBitsClear (
65805|     PRTL_BITMAP BitMapHeader,
65806|     ULONG StartingIndex,
65807|     ULONG Length
65808| );
65809|
65810| NTSYSAPI
65811| BOOLEAN
65812| NTAPI
65813| RtlAreBitsSet (
65814|     PRTL_BITMAP BitMapHeader,
65815|     ULONG StartingIndex,
65816|     ULONG Length
65817| );
65818| #endif
65819|
65820|
65821|
65822| File Listing: SBPSMAN.cpp
65823|
65824| #include "precomp.h"
65825| #include "buildnum.h"
65826|
65827| #ifdef DEBUG
65828| extern "C" void DumpSizeofs(void);
65829| #endif
65830|
65831| #ifdef ALLOC_PRAGMA
65832|     #pragma alloc_text(INIT, DriverEntry)
65833|     #ifdef DEBUG
65834|         #pragma alloc_text(INIT,DumpSizeofs)
65835|     #endif
65836|
65837| #endif
65838|
65839|
65840| #ifdef DEBUG
65841| GLOBALTYPE ULONG DebugLevel=0x00000000;

```

```

65842| GLOBALTYPE ULONG DebugPrints=0;
65843| #endif
65844|
65845| GLOBALTYPE PDRIVER_OBJECT   PSMANDriverObject = NULL;
65846| GLOBALTYPE PDEVICE_OBJECT   PSMANObject=NULL;
65847| #ifdef DEBUG
65848| GLOBALTYPE PSBPSMAN_EXTENSION
        | PSMANObjectExtension=NULL;
65849| #endif
65850| GLOBALTYPE WCHAR
        | gSnapShotDirName[200]={0};
65851| GLOBALTYPE BOOLEAN          PsmActive=FALSE;
65852| GLOBALTYPE BOOLEAN          PSMANPSMInited = FALSE;
65853| GLOBALTYPE KSEMAPHORE       ThreadSemaphore={0};
65854| GLOBALTYPE KEVENT           WorkerThreadEvent={0};
65855| GLOBALTYPE KEVENT           Thread0Inited={0};
65856| GLOBALTYPE KSEMAPHORE       WriteSemaphore={0};
65857| GLOBALTYPE KSEMAPHORE
        | WriteAfterReadSemaphore={0};
65858| GLOBALTYPE KEVENT           PSMANExitingEvent={0};
65859|
65860| GLOBALTYPE ULONG
        | NumberOfThreads=NUMTHREADS_DEF;
65861| GLOBALTYPE ULONG            GlobalThreadCount=0;
65862| GLOBALTYPE PMY_THREAD        ThreadObjects=NULL;
65863| GLOBALTYPE tMY_THREAD        WriteThreadObject={0};
65864|
65865| GLOBALTYPE LIST_ENTRY        ThreadsWorkToDoQueue={0};
65866| GLOBALTYPE KSPIN_LOCK
        | ThreadsWorkToDoSpinLock={0};
65867|
65868| GLOBALTYPE LIST_ENTRY        WriteAfterReadQueue={0};
65869| GLOBALTYPE KSPIN_LOCK
        | WriteAfterReadSpinLock={0};
65870|
65871| GLOBALTYPE LIST_ENTRY        ProcessingQueue={0};
65872| GLOBALTYPE LIST_ENTRY        WriteQueue={0};
65873| GLOBALTYPE KSPIN_LOCK        WriteSpinLock={0};
65874|
65875| GLOBALTYPE ULONG            DoPagingFile=0;
65876| volatile GLOBALTYPE ULONG
        | OutstandingRequests=0;
65877| GLOBALTYPE ULONG            PSMANPSMFlags=0;
65878| GLOBALTYPE FAST_MUTEX        WorkerThreadMutex={0};
65879| GLOBALTYPE FAST_MUTEX        CacheThresholdMutex={0};
        | // Mutex for threshold handling
65880| GLOBALTYPE FAST_MUTEX        PSMUserMutex={0};
65881| GLOBALTYPE KSEMAPHORE
        | PSMOpenCloseSemaphore={0};
65882| #ifdef DEBUG_SNAPSHOTS

```

```

65883| GLOBALTYPE KSEMAPHORE      PSMSnapshotSemaphore;
    | // Semaphore for read/write snapshots
65884| #endif
65885| GLOBALTYPE KSEMAPHORE      PSMVDiskSemaphore={0};
65886| GLOBALTYPE NTSTATUS
    | LastErrorStatus=STATUS_SUCCESS;
65887| GLOBALTYPE FAST_MUTEX      PSMManMemoryMutex={0}; //
    | mutex for memory routines
65888| GLOBALTYPE ULONG           MaxWriteQueueDepth=0;
65889| GLOBALTYPE ULONG           WriteQueueDepth=0;
65890| volatile GLOBALTYPE ULONG   ThreadsAwake=0;
65891| GLOBALTYPE ULONG
    | NewThreadStartDelay=NEWTHREADDELAY_DEF; // in
    | microseconds (1000000 = 1 second)
65892| GLOBALTYPE ULONG
    | MaxThreads=MAXTHREADS_DEF;
65893| GLOBALTYPE ULONG
    | PSMManCreateOptions=CREATEOPTIONS_DEF;
65894| GLOBALTYPE ULONG
    | PSMManOpenOptions=OPENOPTIONS_DEF;
65895| GLOBALTYPE ULONG
    | PSMManFillOnWrite=FILLONWRITES_DEF;
65896| GLOBALTYPE ULONG
    | gHungSystemTimeOut=HUNGSYSTEMTIMEOUT_DEF;
65897| GLOBALTYPE ULONG           gLogErrors=1;
65898| GLOBALTYPE ULONG           gFailFreed=0;
65899| GLOBALTYPE ULONG           gLogOpenClose=1;
65900| #ifdef DEBUG
65901| GLOBALTYPE KSEMAPHORE      PSM_DebugSemaphore={0};
    | // Semaphore to write to log file
65902| GLOBALTYPE ULONG           gDebugToLog=TRUE;
    | // whether to log debug to log file or not
65903| GLOBALTYPE LIST_ENTRY      PSM_DebugQueue={0}; //
    | Queue for threads
65904| GLOBALTYPE KSPIN_LOCK      PSM_DebugSpinLock={0};
    | // spinlock
65905| #endif
65906| GLOBALTYPE ULONG           gVDiskIOHandling=0x0000;
65907|
65908| // from tdrom.c
65909| GLOBALTYPE KEVENT           VDiskExitingEvent={0};
65910|
65911| GLOBALTYPE LIST_ENTRY        ReadVDiskQueue={0};
65912| GLOBALTYPE KSPIN_LOCK        ReadVDiskSpinLock={0};
65913| GLOBALTYPE KSEMAPHORE        ReadVDiskSemaphore={0};
65914|
65915| GLOBALTYPE LIST_ENTRY        WriteVDiskQueue={0};
65916| GLOBALTYPE KSPIN_LOCK        WriteVDiskSpinLock={0};
65917| GLOBALTYPE KSEMAPHORE
    | WriteVDiskSemaphore={0};

```

```

65918|
65919| GLOBALTYPE  ULONG          VDiskNumberOfThreads=0;
65920| GLOBALTYPE  FAST_MUTEX      VDiskThreadMutex={0};
65921|
65922| //GLOBALTYPE  ULONG
        | gNumberOfBuffers=NUMBEROFBUFFERS_DEF;
65923| //GLOBALTYPE  ULONG
        | gBufferSize=BUFFERSIZE_DEF;
65924|
65925| GLOBALTYPE  ULONG
        | gAllowWrites=ALLOWWRITES_DEF;
65926| //GLOBALTYPE  ULONG
        | gWriteCacheMaxSize=WRITECACHEMAXSIZE_DEF;
65927| //GLOBALTYPE  ULONG          gScanLuns=0;
65928|
65929| GLOBALTYPE  UNICODE_STRING    gRegistryPath={0};
65930|
65931| //GLOBALTYPE  struct sCache    VDiskCache={0};
65932| GLOBALTYPE  HANDLE
        | gVDiskRootDirHandle=NULL;
65933| GLOBALTYPE  ULONG GlobalBuildNumber =
        | _BuildNumber_;
65934| GLOBALTYPE  ULONG GlobalVersionMajor =
        | (PSM_CURRENT_VERSION & 0xff00) >> 8;
65935| GLOBALTYPE  ULONG GlobalVersionMinor =
        | PSM_CURRENT_VERSION & 0x00ff;
65936| GLOBALTYPE  PEPROCESS GlobalSystemProcessId=0;
65937|
65938| //GLOBALTYPE  ULONG          gDoSeek=0;
65939| //GLOBALTYPE  ULONG
        | gCacheFlags=CACHEFLAGS_DEF;
65940| // end from tdrom.c
65941|
65942| #ifdef DEBUG
65943| KBUGCHECK_CALLBACK_RECORD BugCheckCallbackRecord={0};
65944| PVOID          BugCheckMemory=NULL;
65945| char            *BugCheckName="PSMAN Bugcheck
        | data";
65946| #endif
65947|
65948| #if _WIN32_WINNT>=0x0500
65949| // {5066C2B1-0F70-4211-B304-68E3629EE386}
65950| DEFINE_GUID(PSManGuid, 0x5066c2b1, 0xf70, 0x4211, 0xb3,
        | 0x4, 0x68, 0xe3, 0x62, 0x9e, 0xe3, 0x86);
65951|
65952| WMIGUIDREGINFO PSManGuidList[] =
65953| {
65954|     { &PSManGuid,
65955|         1,
65956|         0

```

```

65957|    }
65958| };
65959| #endif
65960|
65961| extern "C" {
65962|     extern PSHORT NtBuildNumber;
65963| }
65964|
65965| #ifdef DEBUG
65966| /*-----
    | -----*/
65967| extern "C" void DumpSizeofs(void)
65968| {
65969|     // ntddk.h
65970|     Debug(DEBUG_INIT,("ntddk.h\n"));
65971|     Debug(DEBUG_INIT,("
    | sizeof(LIST_ENTRY)=%d\n",sizeof(LIST_ENTRY)));
65972|     Debug(DEBUG_INIT,("
    | sizeof(ERESOURCE)=%d\n",sizeof(ERESOURCE)));
65973|     Debug(DEBUG_INIT,("
    | sizeof(KSPIN_LOCK)=%d\n",sizeof(KSPIN_LOCK)));
65974|     Debug(DEBUG_INIT,("
    | sizeof(KEVENT)=%d\n",sizeof(KEVENT)));
65975|     Debug(DEBUG_INIT,("
    | sizeof(PARTITION_INFORMATION)=%d\n",sizeof(PARTITION_INF
    | ORMATION)));
65976|     Debug(DEBUG_INIT,("
    | sizeof(KSEMAPHORE)=%d\n",sizeof(KSEMAPHORE)));
65977|     Debug(DEBUG_INIT,("
    | sizeof(FAST_MUTEX)=%d\n",sizeof(FAST_MUTEX)));
65978|     Debug(DEBUG_INIT,("
    | sizeof(RTL_BITMAP)=%d\n",sizeof(RTL_BITMAP)));
65979|     Debug(DEBUG_INIT,("
    | sizeof(FILE_OBJECT)=%d\n",sizeof(FILE_OBJECT)));
65980|     Debug(DEBUG_INIT,("
    | sizeof(DEVICE_OBJECT)=%d\n",sizeof(DEVICE_OBJECT)));
65981|     Debug(DEBUG_INIT,("
    | sizeof(DRIVER_OBJECT)=%d\n",sizeof(DRIVER_OBJECT)));
65982|
65983|     // sbpsman.h
65984|     Debug(DEBUG_INIT,("sbpsman.h\n"));
65985|     Debug(DEBUG_INIT,("
    | sizeof(DEVICE_EXTENSION)=%d\n",sizeof(DEVICE_EXTENSION))
    | );
65986|     Debug(DEBUG_INIT,("
    | sizeof(SBPSMAN_EXTENSION)=%d\n",sizeof(SBPSMAN_EXTENSION
    | ));
65987|     Debug(DEBUG_INIT,("
    | sizeof(FILTERED_EXTENSION)=%d\n",sizeof(FILTERED_EXTENSI
    | ON)));

```



```

65988|   Debug(DEBUG_INIT,("
| sizeof(VDISK_EXTENSION)=%d\n",sizeof(VDISK_EXTENSION)));
65989|   Debug(DEBUG_INIT,("
| sizeof(tMY_THREAD)=%d\n",sizeof(tMY_THREAD)));
65990|   Debug(DEBUG_INIT,("
| sizeof(tPHYSICAL_DEVICE)=%d\n",sizeof(tPHYSICAL_DEVICE))
| );
65991|   Debug(DEBUG_INIT,("
| sizeof(tLOGICAL_DEVICE)=%d\n",sizeof(tLOGICAL_DEVICE)));
65992|   Debug(DEBUG_INIT,("
| sizeof(tOT_USER)=%d\n",sizeof(tOT_USER)));
65993|   Debug(DEBUG_INIT,("
| sizeof(tWriteRequest)=%d\n",sizeof(tWriteRequest)));
65994|   Debug(DEBUG_INIT,("
| sizeof(tReadRequest)=%d\n",sizeof(tReadRequest)));
65995|   Debug(DEBUG_INIT,("
| sizeof(tOpenPsmThread)=%d\n",sizeof(tOpenPsmThread)));
65996|   Debug(DEBUG_INIT,("
| sizeof(tkSnapshotEntry)=%d\n",sizeof(tkSnapshotEntry)));
65997|   Debug(DEBUG_INIT,("
| sizeof(tkSnapshotMaster)=%d\n",sizeof(tkSnapshotMaster))
| );
65998|
65999|   // ioctl.h
66000|   Debug(DEBUG_INIT,("ioctl.h\n"));
66001|   Debug(DEBUG_INIT,("
| sizeof(tPSM_FreeVolume)=%d\n",sizeof(tPSM_FreeVolume)));
66002|   Debug(DEBUG_INIT,("
| sizeof(tOpenTransactionInInternal)=%d\n",sizeof(tOpenTra
| nsactionInInternal)));
66003|
66004|   // rbtree.h
66005|   Debug(DEBUG_INIT,("tree.h\n"));
66006|   Debug(DEBUG_INIT,("
| sizeof(tTreeLeaf)=%d\n",sizeof(tTreeLeaf)));
66007|   Debug(DEBUG_INIT,("
| sizeof(tTree)=%d\n",sizeof(tTree)));
66008|
66009|   // psm.h
66010|   Debug(DEBUG_INIT,("psm.h\n"));
66011|   Debug(DEBUG_INIT,("
| sizeof(tPSM_VersionInfo)=%d\n",sizeof(tPSM_VersionInfo))
| );
66012|   Debug(DEBUG_INIT,("
| sizeof(tOpenTransactionIn)=%d\n",sizeof(tOpenTransaction
| In)));
66013|   Debug(DEBUG_INIT,("
| sizeof(tOpenTransactionOutW)=%d\n",sizeof(tOpenTransacti
| onOutW)));
66014|   Debug(DEBUG_INIT,("

```

```

        | sizeof(tPSM_GetErrorOut)=%d\n",sizeof(tPSM_GetErrorOut))
    | );
66015|     Debug(DEBUG_INIT,("
        | sizeof(tPSM_GetProgressOut)=%d\n",sizeof(tPSM_GetProgres
        | sOut)));
66016|     Debug(DEBUG_INIT,("
        | sizeof(tPSM_RangeList)=%d\n",sizeof(tPSM_RangeList)));
66017|     Debug(DEBUG_INIT,("
        | sizeof(tPSM_FreeRanges)=%d\n",sizeof(tPSM_FreeRanges)));
66018|
66019|     // ondisk.h
66020|     Debug(DEBUG_INIT,("ondisk.h\n"));
66021|     Debug(DEBUG_INIT,("
        | sizeof(tPartRec)=%d\n",sizeof(tPartRec)));
66022|     Debug(DEBUG_INIT,("
        | sizeof(tMBR)=%d\n",sizeof(tMBR)));
66023|     Debug(DEBUG_INIT,("
        | sizeof(NTFS_BOOT_SECTOR)=%d\n",sizeof(NTFS_BOOT_SECTOR))
        | );
66024|     Debug(DEBUG_INIT,("
        | sizeof(FAT_BOOT_SECTOR)=%d\n",sizeof(FAT_BOOT_SECTOR)));
66025|     Debug(DEBUG_INIT,("
        | sizeof(FAT32_BOOT_SECTOR)=%d\n",sizeof(FAT32_BOOT_SECTOR
        | ));
66026|     Debug(DEBUG_INIT,("
        | sizeof(FSINFO_SECTOR)=%d\n",sizeof(FSINFO_SECTOR)));
66027| }
66028| #endif
66029|
66030| VOID
66031| InitAfterFileSys(
66032|     IN PDRIVER_OBJECT DriverObject,
66033|     IN PVOID          Param,
66034|     IN ULONG          Count
66035| )
66036|
66037| {
66038|     return;
66039| }
66040|
66041|
66042| /*-----
        | -----*/
66043|
66044| extern "C"
66045| NTSTATUS
66046| DriverEntry(
66047|     IN PDRIVER_OBJECT DriverObject,
66048|     IN PUNICODE_STRING RegistryPath
66049| )

```

```

66050|
66051| /*++
66052|
66053| Routine Description:
66054|
66055|     This is the routine called by the system to
        | initialize the disk
66056|     performance driver. The driver object is set up and
        | then the
66057|     driver calls PSMANInitialize to attach to the boot
        | devices.
66058|
66059|     IRQL = PASSIVELEVEL
66060| Arguments:
66061|
66062|     DriverObject - The disk performance driver object.
66063|
66064| Return Value:
66065|
66066|     NTSTATUS
66067|
66068| --*/
66069|
66070| {
66071|     NTSTATUS     ntStatus=0;
66072|     WCHAR        deviceNameBuffer[100]={0};
66073|     UNICODE_STRING deviceNameUnicodeString={0};
66074|     WCHAR        deviceLinkBuffer[100]={0};
66075|     UNICODE_STRING deviceLinkUnicodeString={0};
66076|     UNICODE_STRING Uni={0};
66077|     ULONG DispCopy=1;
66078|     ULONG DispNotice=0;
66079|     ULONG LogVersion=1;
66080| #ifdef DEBUG
66081|     ULONG shouldBreak=0;
66082| #endif
66083|
66084|     PAGED_CODE();
66085|
66086| #if TRACK_CONTENTIONS
66087|     KeInitializeSpinLock(&ContentionSpinLock);
66088| #endif
66089|
66090|     MemTrackInit(RegistryPath);
66091|     InitProfiler();
66092|
66093|
        | swprintf(deviceNameBuffer,L"%s_%%04x",SBPSMAN_DEVICE_NAME
        | ,PSM_LOW_COMPATIBLE_VERSION);
66094|

```

```

        | swprintf(deviceLinkBuffer,L"%s_%04x",SBPSMAN_WIN32_NAME,
        | PSM_LOW_COMPATIBLE_VERSION);
66095|
66096|   GlobalSystemProcessId = PsGetCurrentProcess();
66097|
66098| #ifdef DEBUG
66099|   // get Debug info first...
66100|
        | Reg_GetULONGKey(RegistryPath,L"BreakOnEntry",0,&shouldBr
        | eak);
66101|
        | Reg_GetULONGKey(RegistryPath,L"DebugLevel",0x0,&DebugLev
        | el);
66102|
66103|   DbgPrint("Psman: DebugLevel = %08lx,
        | Flags=%08x\n",DebugLevel,DriverObject->Flags);
66104|
66105|   if ( shouldBreak ) {
66106|       DbgPrint("Breaking in Psman driver entry\n");
66107|       DbgBreakPoint();
66108|   }
66109|
66110|
66111|
        | KeInitializeSemaphore(&PSM_DebugSemaphore,0,MAXLONG);
66112|   InitializeListHead(&PSM_DebugQueue);
66113|   KeInitializeSpinLock(&PSM_DebugSpinLock);
66114|
66115|
        | pmRegisterObject(&PSM_DebugSpinLock,"PSM_DebugSpinLock",
        | pmSpinLock);
66116|
        | pmRegisterObject(&ContentionSpinLock,"ContentionSpinLock
        | ",pmSpinLock);
66117|
        | pmRegisterObject(&WriteSpinLock,"WriteSpinLock",pmSpinLo
        | ck);
66118|
66119|
        | pmRegisterObject(&PSM_DebugSemaphore,"PSM_DebugSemaphore
        | ",pmSemaphore);
66120|
        | pmRegisterObject(&PSMOpenCloseSemaphore,"PSMOpenCloseSem
        | aphore",pmSemaphore);
66121|
        | pmRegisterObject(&PSMVDiskSemaphore,"PSMVDiskSemaphore",
        | pmSemaphore);
66122|
        | pmRegisterObject(&WriteVDiskSemaphore,"WriteVDiskSemapho
        | re",pmSemaphore);

```

```

66123|
    | pmRegisterObject(&ReadVDiskSemaphore,"ReadVDiskSemaphore
    | ",pmSemaphore);
66124|
    | pmRegisterObject(&ThreadSemaphore,"ThreadSemaphore",pmSe
    | maphore);
66125|
    | pmRegisterObject(&WriteSemaphore,"WriteSemaphore",pmSema
    | phore);
66126|
    | pmRegisterObject(&WriteAfterReadSemaphore,"WriteAfterRea
    | dSemaphore",pmSemaphore);
66127| #ifdef DEBUG_SNAPSHOTS
66128|
    | pmRegisterObject(&PSMSnapShotSemaphore,"PSMSnapShotSemap
    | hore",pmSemaphore);
66129| #endif
66130|
66131|
    | pmRegisterObject(&WorkerThreadMutex,"WorkerThreadMutex",
    | pmMutex);
66132|
    | pmRegisterObject(&PSMUserMutex,"PSMUserMutex",pmMutex);
66133|
    | pmRegisterObject(&PSManMemoryMutex,"PSManMemoryMutex",pm
    | Mutex);
66134|
    | pmRegisterObject(&VDiskThreadMutex,"VDiskThreadMutex",pm
    | Mutex);
66135|
66136|    DumpSizeofs();
66137| #endif
66138|
66139|
66140|    // write the build number to the registry
66141|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,RegistryPath
    | ->Buffer,L"Build",REG_DWORD,&GlobalBuildNumber,sizeof(DW
    | ORD));
66142|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,RegistryPath
    | ->Buffer,L"VersionHi",REG_DWORD,&GlobalVersionMajor,size
    | of(DWORD));
66143|
    | RtlWriteRegistryValue(RTL_REGISTRY_ABSOLUTE,RegistryPath
    | ->Buffer,L"VersionLo",REG_DWORD,&GlobalVersionMinor,size
    | of(DWORD));
66144|
66145|    // display our copyright
66146|

```

```

    | Reg_GetULONGKey(RegistryPath,L"DisplayCopyright",0,&Disp
    | Copy);
66147|   if ( DispCopy ) {
66148|       WCHAR Buff[80];
66149|       swprintf(Buff,L"Persistent Storage Manager (tm)
    | version %d.%02x build %d
    | %s\n",GlobalVersionMajor,GlobalVersionMinor,GlobalBuildN
    | umber,
66150| #ifdef DEBUG
66151|         L"debug"
66152| #else
66153|         L""
66154| #endif
66155|     );
66156|     RtlInitUnicodeString (&Uni, Buff);
66157|     ZwDisplayString( &Uni );
66158|
66159|     RtlInitUnicodeString (&Uni, L"Copyright (c)
    | 1995-2001 Columbia Data Products, Inc. All Rights
    | Reserved!\n");
66160|     ZwDisplayString( &Uni );
66161|
66162| }
66163|
66164| Debug(DEBUG_INIT,("Persistent Storage Manager (tm)
    | version %d.%02x build
    | %d\n",GlobalVersionMajor,GlobalVersionMinor,GlobalBuildN
    | umber));
66165|
66166|
    | Reg_GetULONGKey(RegistryPath,L"LogVersion",1,&LogVersion
    | );
66167|   if ( LogVersion ) {
66168|       WCHAR VersionHigh[5];
66169|       WCHAR VersionLow[5];
66170|       WCHAR Build[5];
66171|       WCHAR Message[10];
66172|       WCHAR *Strings[4];
66173|       swprintf(VersionHigh,L"%d",GlobalVersionMajor);
66174|
    | swprintf(VersionLow,L"%02x",GlobalVersionMinor);
66175|       swprintf(Build,L"%d",GlobalBuildNumber);
66176|       swprintf(Message,L"%s",
66177| #ifdef DEBUG
66178|         L"debug"
66179| #else
66180|         L""
66181| #endif
66182|     );
66183|       Strings[0] = VersionHigh;

```

```

66184|     Strings[1] = VersionLow;
66185|     Strings[2] = Build;
66186|     Strings[3] = Message;
66187|
66188|     /*lint -save -e740 */
66189|
66190|     | LogError((PDEVICE_OBJECT)DriverObject,NULL,PSM_VERSION_I
66191|     | NFORMATION,0,NULL,0,Strings,4);
66192|
66193|     /*lint -restore */
66194| }
66195|
66196| #if _WIN32_WINNT>=0x0500
66197| // compiling win2k (5.x) version
66198| if ( *NtBuildNumber<=1381 ) {
66199|     /*lint -save -e740 */
66200|
66201|     | LogError((PDEVICE_OBJECT)DriverObject,NULL,PSM_ERROR_WRO
66202|     | NG_VERSION,0,NULL,0,NULL,0);
66203|     /*lint -restore */
66204| #ifdef DEBUG
66205|
66206|     | PManBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,4,ntSt
66207|     | atus,0);
66208| #endif
66209|
66210|     return PSM_ERROR_WRONG_VERSION;
66211| }
66212| #else
66213| // compiling 3.5x/4.x version
66214| if ( *NtBuildNumber>1381 ) {
66215|     /*lint -save -e740 */
66216|
66217|     | LogError((PDEVICE_OBJECT)DriverObject,NULL,PSM_ERROR_WRO
66218|     | NG_VERSION,0,NULL,0,NULL,0);
66219|     /*lint -restore */
66220| #ifdef DEBUG
66221|
66222|     | PManBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,4,ntSt
66223|     | atus,0);
66224| #endif
66225|
66226|     return PSM_ERROR_WRONG_VERSION;
66227| }
66228| #endif
66229|
66230| // display user notice
66231|
66232| | Reg_GetULONGKey(RegistryPath,L"DisplayNotice",0,&DispNot
66233| | ice);
66234| if ( DispNotice ) {
66235|     ULONG i=0;

```

```

66222|     WCHAR Buff[128]={0};
66223|
66224|     | Reg_GetStringKey(RegistryPath,L"Notice",L"Notice",&Uni);
66225|     ZwDisplayString( &Uni );
66226|     Reg_FreeString(&Uni);
66227|     RtlInitUnicodeString (&Uni, L"\n");
66228|     ZwDisplayString( &Uni );
66229|
66230|     for ( i=2;i<=9;i++ ) {
66231|         swprintf(Buff,L"Notice%d",i);
66232|
66233|         | Reg_GetStringKey(RegistryPath,Buff,L"\t",&Uni);
66234|         if ( !((Uni.Buffer[0] == L'\t') &&
66235|         | (Uni.Buffer[1] == 0)) ) {
66236|             ZwDisplayString( &Uni );
66237|             Reg_FreeString(&Uni);
66238|             RtlInitUnicodeString (&Uni, L"\n");
66239|             ZwDisplayString( &Uni );
66240|         } else {
66241|             Reg_FreeString(&Uni);
66242|             break;
66243|         }
66244|     }
66245| // if displayed text, add a extra line between it
66246| | and the devices being PSMed.
66247| if ( (DispCopy) || (DispNotice) ) {
66248|     RtlInitUnicodeString (&Uni, L" \n");
66249|     ZwDisplayString( &Uni );
66250| }
66251| SbGetRegistrySettings( RegistryPath );
66252|
66253| {
66254|     LARGE_INTEGER Perf;
66255|     KeQueryPerformanceCounter( &Perf );
66256|     Debug(DEBUG_INIT,("Performance frequency =
66257|     | %08x%08x\n",Perf.HighPart,Perf.LowPart));
66258| }
66259|
66260| PSManDriverObject = DriverObject;
66261|
66262| // Create an NON EXCLUSIVE device object (multiple
66263| | threads
66264| // can make requests to this device)
66265| RtlInitUnicodeString (&deviceNameUnicodeString,

```



```

66266|         deviceNameBuffer);
66267|
66268|     {
66269| #ifdef DEBUG_EXTENSION
66270|         ULONG SizeOfDeviceExt =
        | sizeof(DEVICE_EXTENSION);
66271| #else
66272|         ULONG SizeOfDeviceExt = SBPSMAN_EXTENSION_SIZE;
66273| #endif
66274|
66275|     #if _WIN32_WINNT >= 0x0500
66276|         #define OTMAN_SECURE_OPTION
        | FILE_DEVICE_SECURE_OPEN
66277|     #else
66278|         #define OTMAN_SECURE_OPTION 0
66279|     #endif
66280|
66281|         ntStatus = IoCreateDevice (DriverObject,
66282|                                     SizeOfDeviceExt,
66283|                                     | &deviceNameUnicodeString,
66284|                                     FILE_DEVICE_UNKNOWN,
66285|                                     OTMAN_SECURE_OPTION,
66286|                                     FALSE,
66287|                                     &PSManObject
66288|                                     );
66289|     #undef OTMAN_SECURE_OPTION
66290| }
66291|
66292| if ( NT_SUCCESS(ntStatus) ) {
66293|     PSBPSMAN_EXTENSION DevExt=NULL;
66294|     ULONG             ulIndex;
66295|     PDRIVER_DISPATCH * dispatch=NULL;
66296|
66297|     Debug(DEBUG_INIT,("Init: Changing security for
        | %p\n",PSManObject));
66298|
66299|     // Fix up the device object's security
        | descriptor by removing
66300|     // access to non-administrators. Don't check
        | the return code because
66301|     // in the bizarre case this would fail, we end
        | up with default security.
66302|     //
66303|     // Here I reach right into the device object to
        | pull out and
66304|     // replace its security descriptor. An
        | alternate way to accomplish
66305|     // the equivalent is to call
        | NtQuerySecurityObject and NtSetSecurityObject.

```

```

66306|      // Both these functions require a handle to the
        | device object, however.
66307|      //
66308|
66309| #if _WIN32_WINNT < 0x0500
66310|      ntStatus = NTIRemoveWorldAce(
        | PSMANObject->SecurityDescriptor,
        | &PSMANObject->SecurityDescriptor );
66311| #endif
66312|
66313|      Debug(DEBUG_INIT,("Init: Internal Device %p
        | created for driver %p\n",PSMANObject,DriverObject));
66314|
66315|      PSMANObject->Flags |= DO_BUFFERED_IO;
66316|
66317| #ifdef DEBUG_EXTENSION
66318|
        | ((PDEVICE_EXTENSION)(PSMANObject->DeviceExtension))->Obj
        | ectType = OBJECT_INTERNAL;
66319|
        | ((PDEVICE_EXTENSION)(PSMANObject->DeviceExtension))->Rea
        | lDeviceExtension =
        | MemAllocatePoolWithTag(NonPagedPool,SBPSMAN_EXTENSION_SI
        | ZE,DEVEXTTAG);
66320|
        | RtlZeroMemory(((PDEVICE_EXTENSION)(PSMANObject->DeviceEx
        | tension))->RealDeviceExtension,SBPSMAN_EXTENSION_SIZE);
66321| #endif
66322|
66323|      DevExt = (PSBPSMAN_EXTENSION)
        | GetDeviceExtension(PSMANObject);
66324|
66325| #ifdef DEBUG
66326|      PSMANObjectExtension=DevExt;
66327| #endif
66328|
66329|      DevExt->DeviceObject = PSMANObject;
66330|      DevExt->DriverObject = DriverObject;
66331|      DevExt->ObjectType = OBJECT_INTERNAL;
66332|      DevExt->LargestBPS = 0;
66333|      DevExt->NumActive = 0;
66334|      DevExt->ShutDownCalled = 0;
66335|      DevExt->PSMUsers = NULL;
66336|
66337|      InitializeListHead(&DevExt->WaitQueue);
66338|
        | KeInitializeSpinLock(&DevExt->WaitQueueSpinLock);
66339|
        | pmRegisterObject(&DevExt->WaitQueueSpinLock,"DevExt->Wai
        | tQueueSpinLock",pmSpinLock);

```

```

66340|
66341|     | ExInitializeResourceLite(&DevExt->DeviceResource);
66342|     | ExInitializeResourceLite(&DevExt->SnapShotResource);
66343|     | pmRegisterObject(&DevExt->DeviceResource,"DevExt->Device
        | Resource",pmRwLock);
66344|     | pmRegisterObject(&DevExt->SnapShotResource,"DevExt->Snap
        | ShotResource",pmRwLock);
66345|
66346|         InitWriteModule();
66347|
66348|         // Set up the device driver entry points.
66349|         Debug(DEBUG_INIT,("Initializing supported
        | functions\n"));
66350|
66351|         //
66352|         // Create dispatch points
66353|         //
66354|         for ( ulIndex = 0, dispatch =
        | DriverObject->MajorFunction;
66355|             ulIndex <= IRP_MJ_MAXIMUM_FUNCTION;
66356|             ulIndex++, dispatch++ ) {
66357|
66358|             // Use PSMANFSPassThru instead of
        | PSMANPassThru so it doesnt
66359|             // modify the OutstandingRequests variable,
        | otherwise we may get an
66360|             // issue when the filesystem filter
        | forwards requests through the
66361|             // system, which goes through the disk
        | driver
66362|             *dispatch = PSMANFSPassThru;
66363|         }
66364|
66365|         DriverObject->MajorFunction[IRP_MJ_CREATE]
        | = PSMANCreate;
66366|         DriverObject->MajorFunction[IRP_MJ_CLOSE]
        | = PSMANClose;
66367|         DriverObject->MajorFunction[IRP_MJ_READ]
        | = PSMANRead;
66368|         DriverObject->MajorFunction[IRP_MJ_WRITE]
        | = PSMANWrite;
66369|
        | DriverObject->MajorFunction[IRP_MJ_FLUSH_BUFFERS]
        | = PSMANFlush;
66370|
        | DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]

```

```

    | = PSMANDeviceControl;
66371|     DriverObject->MajorFunction[IRP_MJ_SHUTDOWN]
    | = PSMANShutdown;
66372|     DriverObject->MajorFunction[IRP_MJ_CLEANUP]
    | = PSMANCleanup;
66373|
66374| #if _WIN32_WINNT >= 0x0500
66375| // we do not support wmi yet. FIXFIXFIX
66376| //
    | DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL] =
    | PSMANWmi;
66377|     DriverObject->MajorFunction[IRP_MJ_PNP]
    | = PSMANPnp;
66378|     DriverObject->MajorFunction[IRP_MJ_POWER]
    | = PSMANPower;
66379|     DriverObject->DriverExtension->AddDevice
    | = PSMANAddDevice;
66380|     DriverObject->DriverUnload
    | = PSMANUnload;
66381|
66382|     Debug(DEBUG_INIT, ("DriverFlags:
    | %08x\n", DriverObject->Flags));
66383| #endif
66384|
66385|     // This is ONLY needed if not a disk driver,
    | iow the disk class driver
66386|     // has already done it for us, and it will call
    | us with shutdown
66387|     // requests. If we wanted to recieve them for
    | our PSMANObject,
66388|     // then we would need call it. Call for every
    | Object you want to
66389|     // recieve shutdown notification.
66390|     IoRegisterShutdownNotification( PSMANObject );
66391|
66392|     // Create a symbolic link, e.g. a name that a
    | Win32 app can specify
66393|     // to open the device
66394|
66395|     RtlInitUnicodeString(&deviceLinkUnicodeString,
66396|                          deviceLinkBuffer);
66397|
66398|     ntStatus = IoCreateSymbolicLink
    | (&deviceLinkUnicodeString,
66399|
    | &deviceNameUnicodeString);
66400|
66401|
66402|     // Call the initialization routine for the
    | first time.

```

```

66403|         if ( NT_SUCCESS(ntStatus) ) {
66404|
66405|             // mutex for user sync
66406|             ExInitializeFastMutex(&PSMUserMutex);
66407|
66408|             // semaphore for Open/close syncing
66409|
66410|             | KeInitializeSemaphore(&PSMOpenCloseSemaphore,1,1);
66411|
66412|             // semaphore for vdisk deletion
66413|
66414|             | KeInitializeSemaphore(&PSMVDiskSemaphore,1,1);
66415|
66416|             #ifdef DEBUG_SNAPSHOTS
66417|             // semaphore for Snapshot
66418|
66419|             | KeInitializeSemaphore(&PSMSnapShotSemaphore,1,1);
66420| #endif
66421|
66422|             // mutex for memory functions
66423|             ExInitializeFastMutex(&PSManMemoryMutex);
66424|
66425|             Debug(DEBUG_INIT | DEBUG_INFO,("Calling
66426|             | InitVDisk function\n"));
66427|             ntStatus =
66428|             | InitVDisk(DriverObject,RegistryPath);
66429|             if ( NT_SUCCESS(ntStatus) ) {
66430|
66431|             #ifdef DEBUG
66432|             // see if we want to display our bug
66433|             | check data.
66434|
66435|             | Reg_GetULONGKey(RegistryPath,L"BugCheckData",0,&DispCopy
66436|             | );
66437|             if ( DispCopy ) {
66438|                 ULONG
66439|                 | MemNeeded=MAX_BACK_LOG_FOR_DEBUG*255+1024;
66440|
66441|                 | KeInitializeCallbackRecord(&BugCheckCallbackRecord);
66442|
66443|                 | BugCheckMemory=MemAllocatePoolWithTag(NonPagedPool,MemNe
66444|                 | eded,BUGCHECK_TAG);
66445|                 if ( BugCheckMemory ) {
66446|                     | KeRegisterBugCheckCallback(&BugCheckCallbackRecord,BugCh
66447|                     | eckCallBack,BugCheckMemory,MemNeeded,(unsigned char
66448|                     | *)BugCheckName);
66449|                 }
66450|             }
66451|
66452|             }
66453|
66454|             }

```

```

66438|
66439|         Reg_GetStringKey (
        | RegistryPath,L"DebugLogFile",L"\t",&Uni);
66440|         if ( !((Uni.Buffer[0] == L'\t') &&
        | (Uni.Buffer[1] == 0)) ) {
66441|             HANDLE TempHandle;
66442|
66443|             Reg_FreeString(&Uni);
66444|
66445|             ntStatus = pmStartThread(
66446|         | (PKSTART_ROUTINE)DebugLogThread, // IN PKSTART_ROUTINE
        | StartRoutine,
66447|             (PVOID)0,
        | // IN PVOID StartContext
66448|             &TempHandle
        | // OUT PHANDLE ThreadHandle,
66449|             );
66450|             if ( NT_SUCCESS(ntStatus) ) {
66451|                 ZwClose(TempHandle);
66452|             } else {
66453|                 ntStatus = 0;
66454|                 goto NoDebugLog;
66455|             }
66456|         } else {
66457|             Reg_FreeString(&Uni);
66458|             NoDebugLog:
66459|
66460|             // free what has been put on list
66461|             {
66462|                 PLIST_ENTRY ListEntry;
66463|                 gDebugToLog= FALSE;
66464|
66465|                 ListEntry =
        | ExInterlockedRemoveHeadList (
66466|         | &PSM_DebugQueue,    // List Head
66467|         | &PSM_DebugSpinLock // Lock
66468|         | );
66469|
66470|                 while ( (ListEntry) &&
        | (ListEntry!=&PSM_DebugQueue) ) {
66471|                     /*lint -save -e413 */
66472|
        | ExFreePool(CONTAINING_RECORD( ListEntry,
        | tDebugLogEntry, ListEntry ));
66473|                     /*lint -restore */
66474|

```

```

66475|             ListEntry =
        | ExInterlockedRemoveHeadList (
66476|
        | &PSM_DebugQueue,    // List Head
66477|
        | &PSM_DebugSpinLock // Lock
66478|
        | );
66479|
66480|         }
66481|     }
66482| }
66483|
66484| #endif
66485|
66486| #if _WIN32_WINNT < 0x0500
66487|     // done for us automatically in win2k
        | drivers
66488|         Debug(DEBUG_INIT | DEBUG_INFO,("Calling
        | Init function\n"));
66489|         PSManInitialize(DriverObject, 0, 0);
66490| #endif
66491|
66492|         ntStatus =
        | FilterDriverEntry(DriverObject,RegistryPath);
66493|         if ( NT_SUCCESS(ntStatus) ) {
66494|
66495|         } else {
66496| #ifdef DEBUG
66497|
        | PSManBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,4,ntSt
        | atus,0);
66498| #endif
66499|
66500|         }
66501|
66502|         } else {
66503| #ifdef DEBUG
66504|
        | PSManBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,3,ntSt
        | atus,0);
66505| #endif
66506|
66507|         }
66508|         } else {
66509| #ifdef DEBUG
66510|
        | PSManBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,2,ntSt
        | atus,0);
66511| #endif

```

```

66512|
66513|     }
66514| } else {
66515| #ifdef DEBUG
66516|     | PSMANBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,1,ntSt
        | atus,0);
66517| #endif
66518| }
66519|
66520| #if _WIN32_WINNT >=0x0500
66521|     // ok register callback so we can be callbacked
        | after the boot time drivers
66522|     // have finished initing (including the filesystem)
66523|
66524|     ULONG DefaultMemSizeInMega = 1;
66525|     ULONG DefaultStepSizeInMega = 1;
66526|
66527|     if(MmIsThisANtAsSystem()) {
66528|         DefaultMemSizeInMega = 10;
66529|         DefaultStepSizeInMega = 2;
66530|     }
66531|
66532|
        | Reg_GetULONGKey(&gRegistryPath,L"NodeInitialSize",Defaul
        | tMemSizeInMega,&DefaultMemSizeInMega);
66533|
        | Reg_GetULONGKey(&gRegistryPath,L"NodeStepSize",DefaultSt
        | epSizeInMega,&DefaultStepSizeInMega);
66534|     ULONG DefaultStepSizeInNodes =
        | (DefaultStepSizeInMega*1024*1024)/sizeof(tTreeLeaf);
66535|
66536|     // always keep one reference count (needed for
        | other components than just the volume
66537|     // trees)
66538|     IncreaseNodeMemory ( DefaultMemSizeInMega,
        | DefaultStepSizeInNodes );
66539| // IoRegisterDriverReinitialization(DriverObject,
        | InitAfterFileSys, (PVOID)0);
66540| #endif
66541|
66542|     // internal system user hasnt been created, lets
        | create it
66543|
        | InternalCreateUser(GlobalSystemProcessId,(_ETHREAD*)-2,N
        | ULL);
66544|
66545|     Debug(DEBUG_INIT,("Returning %08x to
        | NT\n",ntStatus));
66546|

```



```

66547| #ifdef DEBUG
66548|     if ( ntStatus!=0 ) {
66549|         | PManBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,0xa,nt
        | Status,0);
66550|     }
66551| #endif
66552|
66553|     return(ntStatus);
66554|
66555| } // DriverEntry
66556|
66557|
66558| #if _WIN32_WINNT <0x0500
66559| /*
66560|     This routine is called at 2 places
66561|     1. during boot.
66562|     2. when changes have been made to the disk
        | (IOCTL_SET_PARTITION_LAYOUT)
66563|
66564|     We need to add new partitions, and delete old ones.
66565|
66566|     Make sure that the device passed in, is a physical
        | device
66567|
66568| */
66569| /*-----*/
        | -----*/
66570| NTSTATUS PManMakePartitionObjects(
66571|         PDEVICE_OBJECT
        | Partition0,
66572|         BOOLEAN BootTime
66573|     )
66574| {
66575|     PFILTERED_EXTENSION ZeroExt =
        | GetFilteredExtension(Partition0);
66576|     PFILTERED_EXTENSION DevExt=NULL;
66577|     PDEVICE_OBJECT DevObj=NULL;
66578|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
66579|     PFILE_OBJECT fileObject=NULL;
66580|     PDRIVE_LAYOUT_INFORMATION partitionInfo=NULL;
66581|     ULONG partNumber=0;
66582|     CCHAR          ntNameBuffer[64]={0};
66583|     STRING          ntNameString={0};
66584|     BOOLEAN          UpdatingPart;
66585|     BOOLEAN          DisabledPart;
66586|     UNICODE_STRING    ntUnicodeString={0};
66587|
66588|     PAGED_CODE();
66589|

```

```

66590|  ASSERT(ZeroExt->IsPhysical);
66591|
66592|  Debug(DEBUG_INIT,("Getting Partition Info for
    | %08x\n",ZeroExt->TargetDeviceObject));
66593|
66594|  //
66595|  // Read partition table
66596|  //
66597|
66598|  Status = IoReadPartitionTable(
66599|      | ZeroExt->TargetDeviceObject,
66600|          512,
66601|          TRUE,
66602|          &partitionInfo);
66603|
66604|  // if that failed, try sending ioctl to get drive
    | layout.
66605|  if ( (!partitionInfo) || (!NT_SUCCESS(Status)) ) {
66606|      Debug(DEBUG_INIT,("Error! %08x
    | IoReadPartitionTable on %08x\n",Status,Partition0));
66607|      //
66608|      // Allocate buffer for drive layout.
66609|      //
66610|
66611|      // do not use MemAllocatePool, because
    | IoReadPartitionTable above doesnt, and this
66612|      // simplifies the code for deallocating the
    | block of memory
66613|      partitionInfo =
    | (PDRIVE_LAYOUT_INFORMATION)ExAllocatePoolWithTag(PagedPo
    | ol, (128 * sizeof(PARTITION_INFORMATION) + 4),TEMPTAG);
66614|
66615|      if ( !partitionInfo ) {
66616|          Debug(DEBUG_INIT,("Error! Out of
    | memory\n"));
66617|          return STATUS_INSUFFICIENT_RESOURCES;
66618|      }
66619|      // no point in this if no media..
66620|      if ( Status!=STATUS_NO_MEDIA_IN_DEVICE ) {
66621|          Status = Sblo_GetDriveLayout(
    | ZeroExt->TargetDeviceObject,
66622|          | partitionInfo,
66623|              128 *
    | sizeof(PARTITION_INFORMATION) + 4);
66624|      }
66625|  } // if ((!partitionInfo) || (!NT_SUCCESS(Status)))
66626|
66627|  if ( !NT_SUCCESS(Status) ) {

```

```

66628|     Debug(DEBUG_INIT,("Error! %08x sending
| partitioninfo to %08x\n",Status,Partition0));
66629|     // removable drives always create a device
| object...
66630|     partitionInfo->PartitionCount=1;
66631|
| partitionInfo->PartitionEntry[0].StartingOffset.QuadPart
| = 0;
66632|
| partitionInfo->PartitionEntry[0].PartitionLength.QuadPart
| t = 0;
66633|     partitionInfo->PartitionEntry[0].HiddenSectors
| = 0;
66634|     partitionInfo->PartitionEntry[0].PartitionType
| = 0;
66635| }
66636|
66637|     ZeroExt->Physical.Signature =
| partitionInfo->Signature;
66638|
| Debug(DEBUG_INIT,("Signature=%08x\n",ZeroExt->Physical.S
| ignature));
66639|
66640|
66641| // okay, now delete any partitions that may have
| been deleted.
66642| for (
| partNumber=partitionInfo->PartitionCount;partNumber<Zero
| Ext->Physical.LastPartitionNumber;partNumber++ ) {
66643|     WCHAR Name[64];
66644|
| swprintf(Name,L"\\Device\\Harddisk%d\\Partition%d",ZeroE
| xt->DiskNumber,partNumber+1);
66645|     DevObj = GetObjectFromName(Name);
66646|     if ( DevObj ) {
66647|         ULONG DD;
66648|
66649|         DevExt=GetFilteredExtension(DevObj);
66650|
66651|         Debug(DEBUG_INIT,("Logical device %08x
| being deactivated\n",DevObj));
66652|         DevExt->NotActive = 1;
66653|
66654|
| Reg_GetULONGKey(&gRegistryPath,L"LogDrives",1,&DD);
66655|         if ( DD ) {
66656|             WCHAR Drive[5];
66657|             WCHAR Partition[5];
66658|             WCHAR *Strings[2];
66659|

```

```

        | swprintf(Drive,L"%d",DevExt->DiskNumber);
66660|         swprintf(Partition,L"%d",partNumber);
66661|         Strings[0] = Drive;
66662|         Strings[1] = Partition;
66663|
66664|
        | LogError(DevObj,NULL,PSM_DRIVES_DELETED_INFORMATION,0,NU
        | LL,0,Strings,2);
66665|     }
66666| }
66667| }
66668|
66669| // okay we need to either "change" what we know
        | about the device
66670| // or add a device.
66671|
66672| for ( partNumber = 1;
66673|     partNumber <= partitionInfo->PartitionCount;
66674|     partNumber++ ) {
66675|
66676|     // assume we are adding a partition
66677|     UpdatingPart = FALSE;
66678|     DisabledPart = FALSE;
66679|
66680|     Debug(DEBUG_INIT,("Partition
        | %d/%d\n",partNumber,partitionInfo->PartitionCount));
66681|
66682|     //
66683|     // Create device name for partition.
66684|     //
66685|
66686|     sprintf(ntNameBuffer,
66687|         "\\Device\\Harddisk%d\\Partition%d",
66688|         ZeroExt->DiskNumber,
66689|         partNumber);
66690|
66691|     RtlInitAnsiString(&ntNameString, ntNameBuffer);
66692|     RtlAnsiStringToUnicodeString(&ntUnicodeString,
        | &ntNameString, TRUE);
66693|
66694|     if (
        | (DevObj=GetObjectFromName(ntUnicodeString.Buffer))!=NULL
        | ) {
66695|         UpdatingPart=TRUE;
66696|         if (
        | (GetFilteredExtension(DevObj))->NotActive ) {
66697|             DisabledPart = TRUE;
66698|         }
66699|     }
66700|

```

```

66701|         if ( !UpdatingPart ) {
66702|
66703|             // Get target device object so we can see:
66704|             // 1. it exists
66705|             // 2. it is not mounted
66706|
66707|             Status =
        | IoGetDeviceObjectPointer(&ntUnicodeString,
66708|             | FILE_READ_ATTRIBUTES,
66709|             | &fileObject,
66710|                                     &DevObj);
66711|
66712|             if ( !NT_SUCCESS(Status) ) {
66713|                 Debug(DEBUG_INIT,("Error! %08x getting
        | Object pointer for %s\n",Status,ntNameBuffer));
66714|                 RtlFreeUnicodeString(&ntUnicodeString);
66715|                 continue;
66716|             }
66717|
66718|             if ( !DevObj ) {
66719|                 Debug(DEBUG_INIT,("Error! DeviceObject
        | is NULL\n"));
66720|                 RtlFreeUnicodeString(&ntUnicodeString);
66721|                 continue;
66722|             }
66723|
66724|
66725|             //
66726|             // Check if this device is already mounted.
66727|             //
66728|
66729|             if ( !DevObj->Vpb || (DevObj->Vpb->Flags &
        | VPB_MOUNTED) ) {
66730|                 // Can't attach to a device that is
        | already mounted.
66731|
66732|                 Debug(DEBUG_INIT,("%s is already
        | mounted\n",ntNameBuffer));
66733|                 // something must be wrong, we have a
        | mounted volume
66734|                 // but we have no object to it..
66735|                 // if this ever does happen, we will
        | not be psming this partition
66736|                 // FIXFIXFIX
66737|                 Debug(DEBUG_INIT,("No device found for
        | mounted volume\n"));
66738|                 ASSERT(FALSE);
66739|                 ObDereferenceObject(fileObject);

```

```

66740|         fileObject = NULL;
66741|         RtlFreeUnicodeString(&ntUnicodeString);
66742|         continue;
66743|     }
66744|
66745|     // we are now done with deviceobject
    | (always free fileobject when given both)
66746|
66747|     ObDereferenceObject(fileObject);
66748|     fileObject = NULL;
66749|
66750|     //
66751|     // Create device object for this partition.
66752|     //
66753|
66754|     {
66755| #ifdef DEBUG_EXTENSION
66756|         ULONG SizeOfDeviceExt =
    | sizeof(DEVICE_EXTENSION);
66757| #else
66758|         ULONG SizeOfDeviceExt =
    | sizeof(FILTERED_EXTENSION);
66759| #endif
66760|         Status =
    | IoCreateDevice(ZeroExt->DriverObject,
66761|         | SizeOfDeviceExt,
66762|         NULL,
66763|         | FILE_DEVICE_DISK,
66764|         0,
66765|         FALSE,
66766|         &DevObj);
66767|     }
66768|
66769|     if ( !NT_SUCCESS(Status) ) {
66770|         Debug(DEBUG_INIT,("Error! %08x Unable
    | to create device for %s\n",Status,ntNameBuffer));
66771|         RtlFreeUnicodeString(&ntUnicodeString);
66772|         continue;
66773|     }
66774|
66775|     Debug(DEBUG_INIT,("Init: Filtered Device %p
    | created for driver %p
    | '%wZ'\n",DevObj,ZeroExt->DriverObject,&ntUnicodeString))
    | ;
66776|
66777| #ifdef DEBUG_EXTENSION
66778|     | ((PDEVICE_EXTENSION)(DevObj->DeviceExtension))->ObjectTy

```

```

    | pe = OBJECT_FILTEREDDISK;
66779|
    | ((PDEVICE_EXTENSION)(DevObj->DeviceExtension))->RealDevi
    | ceExtension =
    | MemAllocatePoolWithTag(NonPagedPool,sizeof(FILTERED_EXTE
    | NSION),DEVEXTTAG);
66780|
    | RtlZeroMemory(((PDEVICE_EXTENSION)(DevObj->DeviceExtensi
    | on))->RealDeviceExtension,sizeof(FILTERED_EXTENSION));
66781| #endif
66782|
66783|     DevExt = GetFilteredExtension(DevObj);
66784|     ASSERT(DevExt);
66785|
66786|     DevExt->DeviceObject = DevObj;
66787|     DevExt->DiskNumber = ZeroExt->DiskNumber;
66788|     DevExt->DriverObject =
    | ZeroExt->DriverObject;
66789|     DevExt->ObjectType = OBJECT_FILTEREDDISK;
66790|
66791|     Debug(DEBUG_DEVCON,("MakePartitionObjects:
    | directio=%d, Setting to false\n",DevExt->DirectIO));
66792|     DevExt->InLoadUnload = FALSE;
66793|     DevExt->DoDirectIO = FALSE;
66794|     DevExt->IsMounted = FALSE;
66795|     Debug(DEBUG_DEVCON,("Just set IsMounted to
    | FALSE for DevExt=%08x, DevObj=%08x\n",DevExt,DevObj));
66796|     Debug(DEBUG_DEVCON,("direct maps
    | cleared\n"));
66797|
    | RtlZeroMemory(&DevExt->Cache,sizeof(tCacheInfo));
66798|
66799|     DevExt->Cache.HeaderFile.FileHandle =
66800|     DevExt->Cache.IndexFile.FileHandle =
66801|     DevExt->Cache.CacheFile.FileHandle =
    | INVALID_HANDLE_VALUE;
66802|
66803|     DevExt->Cache.HeaderFile.WaitHandle =
66804|     DevExt->Cache.IndexFile.WaitHandle =
66805|     DevExt->Cache.CacheFile.WaitHandle =
    | INVALID_HANDLE_VALUE;
66806|
    | InitializeListHead(&DevExt->Cache.SnapShotHead);
66807|
66808|     // Store pointer to physical device.
66809|     DevExt->PhysicalDevice = Partition0;
66810|     DevExt->IsPhysical = 0;
66811|
66812|     DevExt->ChangeCount =          // for
    | removables.

```

```

66813|
    | DevExt->SignalRead =
66814|
    | DevExt->SignalWrite =
66815|
    | DevExt->PSMed      =
66816|
    | DevExt->NumberOfReadRequests =
66817|
    | DevExt->SectorsRead =
66818|
    | DevExt->NumberOfWriteRequests =
66819|
    | DevExt->SectorsWritten =
66820|
    | DevExt->OpenCount  = 0;      // number of times it
    | has been opened
66821|
66822|         DevExt->DeviceShutDown = 0;
66823|
66824|         // initialize Read and write events
66825|         KeInitializeEvent ( &DevExt->ReadEvent,
66826|                             SynchronizationEvent,
    | // type (notification or sync)
66827|                             FALSE
    | // signaled
66828|                             );
66829|
66830|         KeInitializeEvent ( &DevExt->WriteEvent,
66831|                             SynchronizationEvent,
    | // type (notification or sync)
66832|                             FALSE
    | // signaled
66833|                             );
66834|
66835|         // init spin lock
66836|         KeInitializeSpinLock (
    | &DevExt->StatisticsSpinLock );
66837|
    | pmRegisterObject(&DevExt->StatisticsSpinLock,"DevExt->St
    | atisticsSpinLock",pmSpinLock);
66838|
66839|
    | ExInitializeResourceLite(&DevExt->DeviceExtResource);
66840|
    | pmRegisterObject(&DevExt->DeviceExtResource,"DevExt->Dev
    | iceExtResource",pmRwLock);
66841|         InitializeListHead(&DevExt->SnapShots);
66842|
66843|         //

```



```

66844|          // Attach to the partition. This call links
        | the newly created
66845|          // device to the target device, returning
        | the target device object.
66846|          //
66847|
66848|          Status = IoAttachDevice(DevObj,
66849|                                  &ntUnicodeString,
66850|                                  | &DevExt->TargetDeviceObject);
66851|
66852|          if ( !NT_SUCCESS(Status) ) {
66853|              Debug(DEBUG_INIT,("Error! %08x Unable
        | to attach to %s\n",Status,ntNameBuffer));
66854|              ExDeleteResourceLite(
        | &DevExt->DeviceExtResource );
66855|
        | pmDeRegisterObject(&DevExt->DeviceExtResource);
66856|              IoDeleteDevice(DevObj);
66857|              continue;
66858|          }
66859|
66860|          //
66861|          // Propagate driver's alignment
        | requirements.
66862|          //
66863|
66864|          ASSERT(DevObj);
66865|
66866|          DevObj->Flags |= DO_DIRECT_IO;
66867|
66868|          DevObj->AlignmentRequirement =
        | DevExt->TargetDeviceObject->AlignmentRequirement;
66869|
66870|          DevObj->StackSize      =
        | DevExt->TargetDeviceObject->StackSize+1;
66871|          DevObj->DeviceType     =
        | DevExt->TargetDeviceObject->DeviceType;
66872|          DevObj->Characteristics =
        | DevExt->TargetDeviceObject->Characteristics;
66873|
66874|      } else {
66875|          /*lint -save -e613 */
66876|          DevExt = GetFilteredExtension(DevObj);
66877|          /*lint -restore */
66878|      }
66879|
66880|      //
66881|      // Maintain the last partition number created.
66882|      //

```

```

66883|
66884|     ZeroExt->Physical.LastPartitionNumber =
        | partNumber;
66885|
66886|     // store the name
66887|     wcscpy(DevExt->Name,ntUnicodeString.Buffer);
66888|
66889|     RtlFreeUnicodeString(&ntUnicodeString);
66890|
66891|     DevExt->Pi.StartingOffset =
        | partitionInfo->PartitionEntry[partNumber-1].StartingOffs
        | et;
66892|     DevExt->Pi.PartitionLength =
        | partitionInfo->PartitionEntry[partNumber-1].PartitionLen
        | gth;
66893|     DevExt->Pi.HiddenSectors =
        | partitionInfo->PartitionEntry[partNumber-1].HiddenSector
        | s;
66894|     DevExt->Pi.PartitionType =
        | partitionInfo->PartitionEntry[partNumber-1].PartitionTyp
        | e;
66895|     DevExt->IsPhysical = FALSE;
66896|     DevExt->NotActive = 0;
66897|     DevExt->Cylinders =
        | ZeroExt->Cylinders;
66898|     DevExt->MediaType =
        | ZeroExt->MediaType;
66899|     DevExt->TracksPerCylinder =
        | ZeroExt->TracksPerCylinder;
66900|     DevExt->SectorsPerTrack =
        | ZeroExt->SectorsPerTrack;
66901|     DevExt->BytesPerSector =
        | ZeroExt->BytesPerSector;
66902|
66903|
66904|     Debug(DEBUG_INIT,("%s: %d (%08x):%d (%08x):
        | Offset=%08x%08x, Length=%08x%08x, Hidden=%d,
        | Type=%d\n",
66905|         UpdatingPart ? "Updating" :
        | "Creating",
66906|         DevExt->DiskNumber,
66907|         Partition0,
66908|         partNumber,
66909|         DevObj,
66910|         DevExt->Pi.StartingOffset.HighPart,
66911|         DevExt->Pi.StartingOffset.LowPart,
66912|         DevExt->Pi.PartitionLength.HighPart,

```

```

66913|
| DevExt->Pi.PartitionLength.LowPart,
66914|          DevExt->Pi.HiddenSectors,
66915|          DevExt->Pi.PartitionType
66916|      ));
66917|
66918|      if ( BootTime ) {
66919|          // can only call HalDisplayString during
| boot
66920|          ULONG DD=1;
66921|
66922|
| Reg_GetULONGKey(&gRegistryPath,L"DisplayDrives",0,&DD);
66923|          if ( DD ) {
66924|              WCHAR Buff[256]={0};
66925|              UNICODE_STRING Uni={0};
66926|
66927|              swprintf(Buff,L"Persistent Storage
| Manager activated for drive %d partition
| %d\n",DevExt->DiskNumber,partNumber);
66928|              RtlInitUnicodeString (&Uni, Buff);
66929|
66930|              ZwDisplayString( &Uni );
66931|              //HalDisplayString(Buff);
66932|          }
66933|      }
66934|
66935|      // log to event viewer if attaching to a new
| device
66936|      // or reenabling a disabled device
66937|      if ( (!UpdatingPart) || ((UpdatingPart) &&
| (DisabledPart)) ) {
66938|          ULONG DD=1;
66939|
66940|
| Reg_GetULONGKey(&gRegistryPath,L"LogDrives",1,&DD);
66941|          if ( DD ) {
66942|              WCHAR Drive[5];
66943|              WCHAR Partition[5];
66944|              WCHAR *Strings[2];
66945|
| swprintf(Drive,L"%d",DevExt->DiskNumber);
66946|              swprintf(Partition,L"%d",partNumber);
66947|              Strings[0] = Drive;
66948|              Strings[1] = Partition;
66949|
66950|
| LogError(DevObj,NULL,PSM_DRIVES_INFORMATION,0,NULL,0,Str
| ings,2);
66951|          }

```

```

66952|     }
66953|
66954|     Debug(DEBUG_INIT,("%s successfully
| attached\n",ntNameBuffer));
66955|
66956|     // Clear the device's init flag as per NT DDK
| KB article on creating device
66957|     // objects from a dispatch routine
66958|     // From this point on, we will get Irp's
66959|     if ( !BootTime ) {
66960|         /*lint -save -e613 */
66961|         DevObj->Flags &= ~DO_DEVICE_INITIALIZING;
66962|         /*lint -restore */
66963|     }
66964| }
66965|
66966| ExFreePool(partitionInfo);
66967| return STATUS_SUCCESS;
66968| }
66969|
66970| NTSTATUS AttachToVeritasLDMVolumeW( WCHAR *VolumeName )
66971| {
66972|     NTSTATUS          status;
66973|     PDEVICE_OBJECT     filterDeviceObject;
66974|     PFILTERED_EXTENSION deviceExtension;
66975|     UNICODE_STRING      ntUnicodeString;
66976|     PIRP                irp;
66977|     ULONG               SizeOfDeviceExt;
66978|     PDISK_GEOMETRY       Geometry;
66979|
66980|     PAGED_CODE();
66981|
66982|     //
66983|     // Create a filter device object for this device
| (partition).
66984|     //
66985|
66986|     RtlInitUnicodeString( &ntUnicodeString,
| VolumeName);
66987|     Debug(DEBUG_INIT,("VeritasLDM: '%wZ'\n",
| &ntUnicodeString));
66988|
66989| #ifdef DEBUG_EXTENSION
66990|     SizeOfDeviceExt = sizeof(DEVICE_EXTENSION);
66991| #else
66992|     SizeOfDeviceExt = sizeof(FILTERED_EXTENSION);
66993| #endif
66994|     status = IoCreateDevice(PManDriverObject,
66995|                             SizeOfDeviceExt,
66996|                             NULL,

```

```

66997|          FILE_DEVICE_DISK,
66998|          0,
66999|          FALSE,
67000|          &filterDeviceObject);
67001|
67002|  if ( !NT_SUCCESS(status) ) {
67003|      Debug(DEBUG_INIT,("VeritasLDM: Cannot create
| filterDeviceObject\n"));
67004| #ifdef DEBUG
67005|     | PSMANBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,0x20,0
| ,0);
67006| #endif
67007|     return status;
67008| }
67009|
67010| filterDeviceObject->Flags |= DO_DIRECT_IO;
67011|
67012| #ifdef DEBUG_EXTENSION
67013|     | ((PDEVICE_EXTENSION)(filterDeviceObject->DeviceExtension
| ))->ObjectType = OBJECT_FILTEREDDISK;
67014|     | ((PDEVICE_EXTENSION)(filterDeviceObject->DeviceExtension
| ))->RealDeviceExtension =
| MemAllocatePoolWithTag(NonPagedPool,sizeof(FILTERED_EXTE
| NSION),DEVEXTTAG);
67015|     | RtlZeroMemory(((PDEVICE_EXTENSION)(filterDeviceObject->D
| eviceExtension))->RealDeviceExtension,sizeof(FILTERED_EX
| TENSION));
67016| #endif
67017|     deviceExtension =
| GetFilteredExtension(filterDeviceObject);
67018|
67019|     RtlZeroMemory(deviceExtension,
| FILTERED_EXTENSION_SIZE);
67020|
67021|     deviceExtension->DeviceObject = filterDeviceObject;
67022|     deviceExtension->ObjectType =
| OBJECT_FILTEREDDISK;
67023|     deviceExtension->DiskNumber = -1;
67024|     // until we know better
67025|     deviceExtension->IsPhysical = FALSE;
67026|     deviceExtension->Physical.LastPartitionNumber = 0;
67027|     deviceExtension->DriverObject = PSManDriverObject;
67028|
67029|     Debug(DEBUG_DEVCON,("AttachToVeritasLDMVolumes:
| directio=%d, Setting to false\n",DevExt->DirectIO));
67030|     deviceExtension->InLoadUnload = FALSE;

```

```

67031|   deviceExtension->DoDirectIO = FALSE;
67032|   deviceExtension->IsMounted = FALSE;
67033|   Debug(DEBUG_DEVCON,("AttachToVeritasLDMVolumes:
    | Just set IsMounted to FALSE for DevExt=%08x,
    | DevObj=%08x\n",deviceExtension,filterDeviceObject));
67034|   | RtlZeroMemory(&deviceExtension->Cache,sizeof(tCacheInfo)
    | );
67035|
67036|   deviceExtension->Cache.HeaderFile.FileHandle =
67037|   deviceExtension->Cache.IndexFile.FileHandle =
67038|   deviceExtension->Cache.CacheFile.FileHandle =
    | INVALID_HANDLE_VALUE;
67039|
67040|   deviceExtension->Cache.HeaderFile.WaitHandle =
67041|   deviceExtension->Cache.IndexFile.WaitHandle =
67042|   deviceExtension->Cache.CacheFile.WaitHandle =
    | INVALID_HANDLE_VALUE;
67043|
67044|   ExInitializeResourceLite (
    | &(deviceExtension->Cache.DirectAccessResource) );
67045|   pmRegisterObject (
    | &(deviceExtension->Cache.DirectAccessResource),
    | "DirectAccessResource", pmRwLock );
67046|
67047|   | InitializeListHead(&deviceExtension->Cache.SnapShotHead)
    | ;
67048|
67049|
67050|   deviceExtension->ChangeCount =          // for
    | removables.
67051|
    | deviceExtension->SignalRead =
67052|
    | deviceExtension->SignalWrite =
67053|
    | deviceExtension->PSMed      =
67054|
    | deviceExtension->NumberOfReadRequests =
67055|
    | deviceExtension->SectorsRead =
67056|
    | deviceExtension->NumberOfWriteRequests =
67057|
    | deviceExtension->SectorsWritten =
67058|
    | deviceExtension->OpenCount  = 0;      // number of
    | times it has been opened
67059|

```

```

67060| deviceExtension->DeviceShutDown = 0;
67061|
67062| // init linked list of snapshots.
67063| InitializeListHead(&deviceExtension->SnapShots);
67064|
67065| // initialize Read and write events
67066| KeInitializeEvent ( &deviceExtension->ReadEvent,
| SynchronizationEvent, FALSE );
67067| KeInitializeEvent ( &deviceExtension->WriteEvent,
| SynchronizationEvent, FALSE );
67068|
67069| // Initialize spinlocks
67070|
67071| KeInitializeSpinLock (
| &deviceExtension->StatisticsSpinLock );
67072|
| pmRegisterObject(&deviceExtension->StatisticsSpinLock,"d
| eviceExtension->StatisticsSpinLock",pmSpinLock);
67073|
67074| //
67075| // Attaches the device object to the highest device
| object in the chain and
67076| // return the previously highest device object,
| which is passed to
67077| // IoCallDriver when pass IRPs down the device
| stack
67078| //
67079|
67080|
| wcscpy(deviceExtension->Name,ntUnicodeString.Buffer);
67081|
67082| status = IoAttachDevice(filterDeviceObject,
67083| &ntUnicodeString,
67084|
| &deviceExtension->TargetDeviceObject);
67085|
67086|
67087| if ( !NT_SUCCESS(status) ) {
67088| IoDeleteDevice(filterDeviceObject);
67089| Debug(DEBUG_INIT,("VeritasLDM: Unable to attach
| %X to target '%S'\n",filterDeviceObject, VolumeName));
67090| #ifdef DEBUG
67091|
| PSMANBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,0x21,0
| ,0);
67092| #endif
67093| return STATUS_NO_SUCH_DEVICE;
67094| }
67095|
67096| //

```

```

67097| // Save the filter device object in the device
      | extension
67098| //
67099| deviceExtension->DeviceObject = filterDeviceObject;
67100|
67101|
      | ExInitializeResourceLite(&deviceExtension->DeviceExtReso
      | urce);
67102|
      | pmRegisterObject(&deviceExtension->DeviceExtResource,"de
      | viceExtension->DeviceExtResource",pmRwLock);
67103|
67104| Debug(DEBUG_INIT,("Getting geometry\n"));
67105| Geometry = (PDISK_GEOMETRY)
      | MemAllocatePoolWithTag(PagedPool,sizeof(DISK_GEOMETRY),T
      | EMPTAG);
67106|
67107| if ( Geometry ) {
67108|
67109|     status =
      | Sblo_GetGeometry(deviceExtension->TargetDeviceObject,Geo
      | metry);
67110|
67111|     if ( !NT_SUCCESS(status) ) {
67112|         Debug(DEBUG_INIT,("Error! %08x sending
      | disk_geometry to
      | '%S'\n",status,deviceExtension->Name));
67113|         // keep going, incase there is no cartridge
      | in drive.
67114|         Geometry->Cylinders.QuadPart = 0;
67115|         Geometry->MediaType          =
      | RemovableMedia;
67116|         Geometry->TracksPerCylinder = 0;
67117|         Geometry->SectorsPerTrack   = 0;
67118|         Geometry->BytesPerSector     = 512;
67119|         status = 0;
67120|     }
67121|
67122| // get the disk geometry
67123| deviceExtension->Cylinders          =
      | Geometry->Cylinders;
67124| deviceExtension->MediaType          =
      | Geometry->MediaType;
67125| deviceExtension->TracksPerCylinder =
      | Geometry->TracksPerCylinder;
67126| deviceExtension->SectorsPerTrack    =
      | Geometry->SectorsPerTrack;
67127| deviceExtension->BytesPerSector     =
      | Geometry->BytesPerSector;
67128|

```



```

67129|    // save largest BPS request
67130|    if ( deviceExtension->BytesPerSector >
    | GlobalData->LargestBPS ) {
67131|        GlobalData->LargestBPS =
    | deviceExtension->BytesPerSector;
67132|    }
67133|
67134|
67135|    Debug(DEBUG_INIT,("Device='%S', Cyls=%d,
    | Heads=%d, SPT=%d, BPS=%d\n",
67136|        deviceExtension->Name,
67137|        Geometry->Cylinders.LowPart,
67138|        Geometry->TracksPerCylinder,
67139|        Geometry->SectorsPerTrack,
67140|        Geometry->BytesPerSector
67141|    ));
67142|
67143|    FREE_POINTER(Geometry);
67144| } else {
67145|     Debug(DEBUG_INIT,("Error! Out of memory\n"));
67146|     deviceExtension->Cylinders.QuadPart
    | = 0;
67147|     deviceExtension->MediaType      =
    | FixedMedia;
67148|     deviceExtension->TracksPerCylinder  = 0;
67149|     deviceExtension->SectorsPerTrack    = 0;
67150|     deviceExtension->BytesPerSector     = 512;
67151| }
67152|
67153|
67154| //
67155| // Clear the DO_DEVICE_INITIALIZING flag
67156| //
67157|
67158| filterDeviceObject->Flags &=
    | ~DO_DEVICE_INITIALIZING;
67159|
67160| Debug(DEBUG_INIT,("VeritasLDM: FilterObject %X
    | attached to '%S'\n",filterDeviceObject,VolumeName));
67161|
67162| return STATUS_SUCCESS;
67163| }
67164|
67165| NTSTATUS AttachToVeritasLDMVolumeA( char *VolumeName )
67166| {
67167|     STRING          ntNameString;
67168|     UNICODE_STRING  ntUnicodeString;
67169|     NTSTATUS        Status;
67170|
67171|     RtlInitAnsiString(&ntNameString, VolumeName);

```

```

67172|   RtlAnsiStringToUnicodeString(&ntUnicodeString,
    | &ntNameString, TRUE);
67173|
67174|   Status =
    | AttachToVeritasLDMVolumeW(ntUnicodeString.Buffer);
67175|
67176|   RtlFreeUnicodeString(&ntUnicodeString);
67177|   return Status;
67178| }
67179|
67180|
67181| NTSTATUS AttachToVeritasLDMVolumeByObject(
    | PDEVICE_OBJECT DeviceObject )
67182| {
67183|   NTSTATUS Status=STATUS_UNSUCCESSFUL;
67184|   POBJECT_NAME_INFORMATION OBI=NULL;
67185|   ULONG Returned=0;
67186|
67187|   // get buffer length need for name
67188|   Status = ObQueryNameString( DeviceObject, NULL, 0,
    | &Returned );
67189|
67190|   if ( (Status==STATUS_INFO_LENGTH_MISMATCH) &&
    | (Returned>0) ) {
67191|
67192|       // allocate memory for it
67193|       OBI = (POBJECT_NAME_INFORMATION)
    | MemAllocatePoolWithTag(PagedPool,Returned,FILENAMETAG);
67194|       if ( OBI ) {
67195|           // get name
67196|           Status = ObQueryNameString( DeviceObject,
    | OBI, Returned, &Returned );
67197|
67198|           if ( NT_SUCCESS(Status) ) {
67199|               Status =
    | AttachToVeritasLDMVolumeW(OBI->Name.Buffer);
67200|               if ( !NT_SUCCESS(Status) ) {
67201|                   Debug(DEBUG_INIT,("Error %08x
    | attaching to veritas LDM '%s'\n",Status,&OBI->Name));
67202|               }
67203|           } else {
67204|               Debug(DEBUG_INIT,("Error %08x
    | ObQueryNameString2 returned %08x\n",Status,Returned));
67205|           }
67206|           FREE_POINTER(OBI);
67207|       } else {
67208|           Debug(DEBUG_INIT,("Out of memory for Object
    | name\n"));
67209|           Status = STATUS_INSUFFICIENT_RESOURCES;
67210|       }

```

```

67211| } else {
67212|     Debug(DEBUG_INIT,("Error %08x ObQueryNameString
| returned %08x\n",Status,Returned));
67213| }
67214| return Status;
67215| }
67216|
67217| PDEVICE_OBJECT GetOurObjectFromTargetObject(
| PDEVICE_OBJECT TargetObject )
67218| {
67219|     PDEVICE_OBJECT DevObj =
| PSMAN_DRIVER_OBJECT->DeviceObject;
67220|
67221|     while ( DevObj ) {
67222|         if (
| PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
67223|             PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(DevObj);
67224|
67225|             if ( DevExt->TargetDeviceObject ==
| TargetObject ) {
67226|                 return DevObj;
67227|             }
67228|         }
67229|
67230|         DevObj = DevObj->NextDevice;
67231|     }
67232|
67233|     Debug(DEBUG_INIT,("Target object %08x not
| found\n",TargetObject));
67234|     return NULL;
67235|
67236| }
67237|
67238| NTSTATUS DetachFromVeritasLDMVolumeByObject(
| PDEVICE_OBJECT DeviceObject )
67239| {
67240|     PDEVICE_OBJECT DevObj =
| GetOurObjectFromTargetObject(DeviceObject);
67241|     PFILTERED_EXTENSION
| DevExt=GetFilteredExtension(DevObj);
67242|
67243|     Debug(DEBUG_INIT,("Detaching %08x from device
| %08x\n",DevObj,DevExt->TargetDeviceObject));
67244|     IoDetachDevice( DevExt->TargetDeviceObject );
67245|
67246|     Debug(DEBUG_INIT,("Deleting Device object\n"));
67247|     ExDeleteResourceLite( &DevExt->DeviceExtResource);
67248|     pmDeRegisterObject(&DevExt->DeviceExtResource);
67249|

```

```

67250|   IoDeleteDevice( DevObj );
67251|   return STATUS_SUCCESS;
67252| }
67253|
67254| void VeritasLDMCreateCallBack( PDEVICE_OBJECT
    | DeviceObject )
67255| {
67256|   NTSTATUS Status = AttachToVeritasLDMVolumeByObject(
    | DeviceObject );
67257|   if ( NT_SUCCESS(Status) ) {
67258|       Debug(DEBUG_INIT,("VeritasLDM: Success
    | attaching to %08x\n",DeviceObject));
67259|   } else {
67260|       Debug(DEBUG_INIT,("VeritasLDM: Unable to attach
    | %08x to %08x\n",Status,DeviceObject));
67261|   }
67262| }
67263|
67264| void VeritasLDMDeleteCallBack( PDEVICE_OBJECT
    | DeviceObject )
67265| {
67266|   NTSTATUS Status =
    | DetachFromVeritasLDMVolumeByObject( DeviceObject );
67267|   if ( NT_SUCCESS(Status) ) {
67268|       Debug(DEBUG_INIT,("VeritasLDM: Success
    | attaching to %08x\n",DeviceObject));
67269|   } else {
67270|       Debug(DEBUG_INIT,("VeritasLDM: Unable to detach
    | %08x to %08x\n",Status,DeviceObject));
67271|   }
67272| }
67273|
67274|
67275| NTSTATUS RegisterVeritasLDMCallbacks( )
67276| {
67277|   NTSTATUS Status=STATUS_SUCCESS;
67278|   struct volk_PSM_callbacks CallBacks;
67279|   HANDLE Handle;
67280|
67281|   // Open the Volume manager info device object
67282|   Status = vol_devfile_open(&Handle,
    | VOL_INFO_DEV_NAME);
67283|
67284|   if ( NT_SUCCESS(Status) ) {
67285|       Debug(DEBUG_INFO,("VeritasLDM:
    | IsInstalled\n"));
67286|
67287|       // get the number of volumes
67288|       CallBacks.PSM_create_callback =
    | VeritasLDMCreateCallBack;

```

```

67289|     Callbacks.PSM_delete_callback =
        | VeritasLDMDeleteCallback;
67290|
67291|     Status = vold_driver_ioctl(Handle,
        | VOL_SET_PSM_CALLBACKS, &Callbacks);
67292|
67293|     if ( NT_SUCCESS(Status) ) {
67294|         Debug(DEBUG_INFO,("VeritasLDM: Success
        | registering callBacks\n"));
67295|     } else {
67296|         Debug(DEBUG_INFO,("VeritasLDM: Error %08x
        | sending ioctl\n",Status));
67297|     }
67298|
67299|     ZwClose(Handle);
67300| } else {
67301|     Debug(DEBUG_INFO,("VeritasLDM: Not
        | installed\n"));
67302| }
67303| return Status;
67304| }
67305|
67306| NTSTATUS ScanForVeritasLDMVolumes()
67307| {
67308|     NTSTATUS Status=0;
67309|     struct volkdevnum_dump kdevn={0};
67310|     ULONG i=0;
67311|     HANDLE Handle=0;
67312|
67313|     // Open the Volume manager info device object
67314|     Status = vold_devfile_open(&Handle,
        | VOL_INFO_DEV_NAME);
67315|
67316|     if ( NT_SUCCESS(Status) ) {
67317|         Debug(DEBUG_INFO,("VeritasLDM:
        | IsInstalled\n"));
67318|
67319|         // get the number of volumes
67320|         Status = vold_driver_ioctl(Handle,
        | VOL_GET_VDEV_VOLNUMS, &kdevn);
67321|
67322|         if ( NT_SUCCESS(Status) ) {
67323|
67324|             Debug(DEBUG_INFO,("VeritasLDM: %08x volumes
        | configured\n",kdevn.kvdev_cnt));
67325|
67326|             // allocate a buffer and get the volumes
67327|             if ( kdevn.kvdev_cnt ) {
67328|                 kdevn.kvdev_name = (char *)
        | MemAllocatePoolWithTag(PagedPool,kdevn.kvdev_cnt *

```

```

    | 128,TEMPTAG);
67329|
67330|         if ( kdevn.kvdev_name ) {
67331|             Status = vold_driver_ioctl(Handle,
    | VOL_GET_VDEV_VOLNUMS, &kdevn);
67332|
67333|             if ( NT_SUCCESS(Status) ) {
67334|                 for ( i = 0; i <
    | kdevn.kvdev_cnt; i++ ) {
67335|                     // dev
    | '\Device\HarddiskDmVolumes\DynamicGroup\Volume1'
67336|
    | Debug(DEBUG_INFO,("VeritasLDM: '%s'\n",
    | &kdevn.kvdev_name[i*128]));
67337|             Status =
    | AttachToVeritasLDMVolumeA(&kdevn.kvdev_name[i*128]);
67338|             if ( !NT_SUCCESS(Status) )
    | {
67339|
    | Debug(DEBUG_INFO,("VeritasLDM: Error %08x attaching to
    | veritas LDM volume\n",Status));
67340|             }
67341|         }
67342|     } else {
67343|         Debug(DEBUG_INFO,("VeritasLDM:
    | Error %08x sending ioctl 2\n",Status));
67344|     }
67345|
67346|         MemFreePool(kdevn.kvdev_name);
67347|     } else {
67348|         Status =
    | STATUS_INSUFFICIENT_RESOURCES;
67349|         Debug(DEBUG_INFO,("VeritasLDM:
    | Error %08x getting memory\n",Status));
67350|     }
67351| } else {
67352|     Debug(DEBUG_INFO,("VeritasLDM: No
    | volumes configured\n"));
67353| }
67354| } else {
67355|     Debug(DEBUG_INFO,("VeritasLDM: Error %08x
    | sending ioctl\n",Status));
67356| }
67357|
67358|     ZwClose(Handle);
67359| } else {
67360|     Debug(DEBUG_INFO,("VeritasLDM: Not
    | installed\n"));
67361| }
67362|

```

```

67363|   return Status;
67364| }
67365|
67366|
67367| /*-----
    | -----*/
67368| VOID
67369| PSMInitialize(
67370|     IN PDRIVER_OBJECT DriverObject,
67371|     IN PVOID          NextDisk,
67372|     IN ULONG          Count
67373| )
67374|
67375| /*++
67376|
67377| Routine Description:
67378|
67379|   Attach to new disk devices and partitions.
67380|   Set up device objects for counts and times.
67381|   If this is the first time this routine is called,
67382|   then register with the IO system to be called
67383|   after all other disk device drivers have initiated.
67384|
67385| Arguments:
67386|
67387|   DriverObject - Disk performance driver object.
67388|   NextDisk - Starting disk for this part of the
    | initialization.
67389|   Count - Not used. Number of times this routine has
    | been called.
67390|
67391| Return Value:
67392|
67393|   NTSTATUS
67394|
67395| --*/
67396|
67397| {
67398|   PCONFIGURATION_INFORMATION
    | configurationInformation=NULL;
67399|   CCHAR          ntNameBuffer[64]={0};
67400|   STRING          ntNameString={0};
67401|   UNICODE_STRING  ntUnicodeString={0};
67402|   PDEVICE_OBJECT  physicalDevice=NULL;
67403|   PFILTERED_EXTENSION  zeroExtension=NULL;
67404|   PDISK_GEOMETRY  Geometry=NULL;
67405|   NTSTATUS        status=0;
67406|   ULONG           diskNumber=0;
67407|   PSBPSMAN_EXTENSION
    | SbotExt=(PSBPSMAN_EXTENSION)GetDeviceExtension(PSManObje

```

```

    | ct);
67408|
67409|     PAGED_CODE();
67410|
67411|     Debug(DEBUG_PROCCALL,("PSManInitialize Called,
    | ND=%ld, Count=%ld\n", (ULONG)NextDisk, Count));
67412|
67413|
67414|     //
67415|     // Get the configuration information.
67416|     //
67417|
67418|     configurationInformation =
    | IoGetConfigurationInformation();
67419|
67420|     //
67421|     // Find disk devices.
67422|     //
67423|
67424|     for ( diskNumber = (ULONG)NextDisk;
67425|         diskNumber <
    | configurationInformation->DiskCount;
67426|         diskNumber++ ) {
67427|
67428|         Debug(DEBUG_INIT,("On Disk
    | %d/%d\n", diskNumber, configurationInformation->DiskCount)
    | );
67429|         //
67430|         // Create device name for the physical disk.
67431|         //
67432|
67433|         sprintf(ntNameBuffer,
67434|             "\\Device\\Harddisk%d\\Partition0",
67435|             diskNumber);
67436|
67437|         RtlInitAnsiString(&ntNameString,
67438|             ntNameBuffer);
67439|
67440|         RtlAnsiStringToUnicodeString(&ntUnicodeString,
67441|             &ntNameString,
67442|             TRUE);
67443|
67444|         //
67445|         // Create device object for partition 0.
67446|         //
67447|
67448|         {
67449| #ifdef DEBUG_EXTENSION
67450|             ULONG SizeOfDeviceExt =
    | sizeof(DEVICE_EXTENSION);

```



```

67451| #else
67452|         ULONG SizeOfDeviceExt =
        | sizeof(FILTERED_EXTENSION);
67453| #endif
67454|         status = IoCreateDevice(DriverObject,
67455|                                 SizeOfDeviceExt,
67456|                                 NULL,
67457|                                 FILE_DEVICE_DISK,
67458|                                 0,
67459|                                 FALSE,
67460|                                 &physicalDevice);
67461|     }
67462|
67463|     if ( !NT_SUCCESS(status) ) {
67464|         Debug(DEBUG_INIT,("Error! %08x creating
        | device for %s\n",status,ntNameBuffer));
67465|         continue;
67466|     }
67467|
67468|     if ( !physicalDevice ) {
67469|         Debug(DEBUG_INIT,("Error! physicalDevice is
        | NULL\n"));
67470|         continue;
67471|     }
67472|
67473|     Debug(DEBUG_INIT,("Init: Filtered Device %p
        | created for driver %p
        | '%wZ'\n",physicalDevice,DriverObject,&ntUnicodeString));
67474|     //
67475|     // Point device extension back at device object
        | and remember
67476|     // the disk number.
67477|     //
67478|
67479| #ifdef DEBUG_EXTENSION
67480|
        | ((PDEVICE_EXTENSION)(physicalDevice->DeviceExtension))->
        | ObjectType = OBJECT_FILTEREDDISK;
67481|
        | ((PDEVICE_EXTENSION)(physicalDevice->DeviceExtension))->
        | RealDeviceExtension =
        | MemAllocatePoolWithTag(NonPagedPool,sizeof(FILTERED_EXTE
        | NSION),DEVEXTTAG);
67482|
        | RtlZeroMemory(((PDEVICE_EXTENSION)(physicalDevice->Devic
        | eExtension))->RealDeviceExtension,sizeof(FILTERED_EXTENS
        | ION));
67483| #endif
67484|
67485|     zeroExtension =

```

```

    | GetFilteredExtension(physicalDevice);
67486|
67487|    // store the name
67488|
    | wcscpy(zeroExtension->Name,ntUnicodeString.Buffer);
67489|
67490|    zeroExtension->DeviceObject = physicalDevice;
67491|    zeroExtension->ObjectType =
    | OBJECT_FILTEREDDISK;
67492|    zeroExtension->DiskNumber = diskNumber;
67493|    zeroExtension->IsPhysical = TRUE;
67494|    zeroExtension->Physical.LastPartitionNumber =
    | 0;
67495|    zeroExtension->DriverObject = DriverObject;
67496|
67497|    Debug(DEBUG_DEVCON,("PSManInitialize:
    | directio=%d, Setting to false\n",DevExt->DirectIO));
67498|    zeroExtension->InLoadUnload = FALSE;
67499|    zeroExtension->DoDirectIO = FALSE;
67500|    zeroExtension->IsMounted = FALSE;
67501|    Debug(DEBUG_DEVCON,("PSManInitialize: just set
    | IsMounted to FALSE for DevExt=%08x,
    | DevObj=%08x\n",DevExt,physicalDevice));
67502|    Debug(DEBUG_DEVCON,("direct maps cleared\n"));
67503|
    | RtlZeroMemory(&zeroExtension->Cache,sizeof(tCacheInfo));
67504|
67505|    zeroExtension->Cache.HeaderFile.FileHandle =
67506|    zeroExtension->Cache.IndexFile.FileHandle =
67507|    zeroExtension->Cache.CacheFile.FileHandle =
    | INVALID_HANDLE_VALUE;
67508|
67509|    zeroExtension->Cache.HeaderFile.WaitHandle =
67510|    zeroExtension->Cache.IndexFile.WaitHandle =
67511|    zeroExtension->Cache.CacheFile.WaitHandle =
    | INVALID_HANDLE_VALUE;
67512|
67513|    ExInitializeResourceLite (
    | &(zeroExtension->Cache.DirectAccessResource) );
67514|    pmRegisterObject (
    | &(zeroExtension->Cache.DirectAccessResource),
    | "DirectAccessResource", pmRwLock );
67515|
67516|
67517|
    | InitializeListHead(&zeroExtension->Cache.SnapShotHead);
67518|
67519|
67520|    zeroExtension->ChangeCount =        //
    | for removables.

```

```

67521|
    | zeroExtension->SignalRead =
67522|
    | zeroExtension->SignalWrite =
67523|
    | zeroExtension->PSMed      =
67524|
    | zeroExtension->NumberOfReadRequests =
67525|
    | zeroExtension->SectorsRead =
67526|
    | zeroExtension->NumberOfWriteRequests =
67527|
    | zeroExtension->SectorsWritten =
67528|
    | zeroExtension->OpenCount  = 0;      // number of
    | times it has been opened
67529|
67530|    zeroExtension->DeviceShutDown = 0;
67531|
67532|    // resource for rbtree functions
67533|    // Supposed to call ExDeleteResourceLite()
    | before
67534|    // freeing mem, but since we never free, lets
    | not worry
67535|    // about it until we add support for unloading.
67536|
67537|
    | ExInitializeResourceLite(&zeroExtension->DeviceExtResour
    | ce);
67538|
    | pmRegisterObject(&zeroExtension->DeviceExtResource,"zero
    | Extension->DeviceExtResource",pmRwLock);
67539|    // init linked list of snapshots.
67540|    InitializeListHead(&zeroExtension->SnapShots);
67541|
67542|    // This is the physical device object.
67543|    zeroExtension->PhysicalDevice = physicalDevice;
67544|
67545|    // Attach to partition0. This call links the
    | newly created
67546|    // device to the target device, returning the
    | target device object.
67547|    status = IoAttachDevice(physicalDevice,
67548|                           &ntUnicodeString,
67549|                           | &zeroExtension->TargetDeviceObject);
67550|
67551|    if ( !NT_SUCCESS(status) ) {
67552|        Debug(DEBUG_INIT,("Error! %08x attaching to

```

```

    | device %s\n",status,ntNameBuffer));
67553|         ExDeleteResourceLite(
    | &zeroExtension->DeviceExtResource );
67554|
    | pmDeRegisterObject(&zeroExtension->DeviceExtResource);
67555|         IoDeleteDevice(physicalDevice);
67556|         continue;
67557|     }
67558|
67559|     RtlFreeUnicodeString(&ntUnicodeString);
67560|
67561|     // Propagate driver's alignment requirements.
67562|     physicalDevice->Flags |= DO_DIRECT_IO;
67563|
67564|     physicalDevice->AlignmentRequirement =
    | zeroExtension->TargetDeviceObject->AlignmentRequirement;
67565|
67566|     physicalDevice->StackSize =
    | zeroExtension->TargetDeviceObject->StackSize+1;
67567|     physicalDevice->DeviceType =
    | zeroExtension->TargetDeviceObject->DeviceType;
67568|     physicalDevice->Characteristics =
    | zeroExtension->TargetDeviceObject->Characteristics;
67569|
67570|     // initialize Read and write events
67571|     KeInitializeEvent ( &zeroExtension->ReadEvent,
67572|         SynchronizationEvent, //
    | type (notification or sync)
67573|         FALSE //
    | signaled
67574|     );
67575|
67576|     KeInitializeEvent ( &zeroExtension->WriteEvent,
67577|         SynchronizationEvent, //
    | type (notification or sync)
67578|         FALSE //
    | signaled
67579|     );
67580|
67581|
67582|
67583|     //
67584|     // Initialize spinlocks
67585|     //
67586|
67587|     KeInitializeSpinLock (
    | &zeroExtension->StatisticsSpinLock );
67588|
    | pmRegisterObject(&zeroExtension->StatisticsSpinLock,"zer
    | oExtension->StatisticsSpinLock",pmSpinLock);

```

```

67589|
67590| /*****
| *****/
67591| //
67592| // Allocate buffer for disk geometry.
67593| //
67594|
67595|     Debug(DEBUG_INIT,("Getting geometry\n"));
67596|     Geometry = (PDISK_GEOMETRY)
| MemAllocatePoolWithTag(PagedPool,sizeof(DISK_GEOMETRY),T
| EMPTAG);
67597|
67598|     if ( !Geometry ) {
67599|         Debug(DEBUG_INIT,("Error! Out of
| memory\n"));
67600|         continue;
67601|     }
67602|
67603|     status =
| Sblo_GetGeometry(zeroExtension->TargetDeviceObject,Geome
| try);
67604|
67605|     if ( !NT_SUCCESS(status) ) {
67606|         Debug(DEBUG_INIT,("Error! %08x sending
| disk_geometry to %s\n",status,ntNameBuffer));
67607|         // keep going, incase there is no cartridge
| in drive.
67608|         Geometry->Cylinders.QuadPart = 0;
67609|         Geometry->MediaType =
| RemovableMedia;
67610|         Geometry->TracksPerCylinder = 0;
67611|         Geometry->SectorsPerTrack = 0;
67612|         Geometry->BytesPerSector = 512;
67613|     }
67614|
67615|     // get the disk geometry
67616|     zeroExtension->Cylinders =
| Geometry->Cylinders;
67617|     zeroExtension->MediaType =
| Geometry->MediaType;
67618|     zeroExtension->TracksPerCylinder =
| Geometry->TracksPerCylinder;
67619|     zeroExtension->SectorsPerTrack =
| Geometry->SectorsPerTrack;
67620|     zeroExtension->BytesPerSector =
| Geometry->BytesPerSector;
67621|
67622|     zeroExtension->IsPhysical = 1;
67623|
67624|     // save largest BPS request

```

```

67625|         if ( zeroExtension->BytesPerSector >
| SbotExt->LargestBPS ) {
67626|             SbotExt->LargestBPS =
| zeroExtension->BytesPerSector;
67627|         }
67628|
67629|
67630|         Debug(DEBUG_INIT,("Device=%p, Cyls=%d,
| Heads=%d, SPT=%d, BPS=%d\n",
67631|             physicalDevice,
67632|             Geometry->Cylinders.LowPart,
67633|             Geometry->TracksPerCylinder,
67634|             Geometry->SectorsPerTrack,
67635|             Geometry->BytesPerSector
67636|             ));
67637|
67638|         FREE_POINTER(Geometry);
67639|
67640| /*****
| *****/
67641|
67642|         // make partitions for this physical device.
67643|         PSMANMakePartitionObjects( physicalDevice, TRUE
| );
67644|     }
67645|
67646|     RegisterVeritasLDMCallbacks();
67647|
67648|     // scan for Veritas Logical Disk Manager volumes
67649|     ScanForVeritasLDMVolumes();
67650|
67651|     //
67652|     // Check if this is the first time this routine has
| been called.
67653|     //
67654|
67655|     if ( !NextDisk ) {
67656|
67657|         //
67658|         // Register with IO system to be called a
| second time after all
67659|         // other device drivers have initialized.
67660|         //
67661|
67662|         IoRegisterDriverReinitialization(DriverObject,
67663|             | PSMANInitialize,
67664|             | (PVOID)configurationInformation->DiskCount);
67665|     }

```

```

67666|
67667|   Debug(DEBUG_PROCCALL,("PSManInitialize Done\n"));
67668|   return;
67669|
67670| } // end PSManInitialize()
67671| #endif // _WIN32_WINNT < 0x0500
67672|
67673|
67674| GLOBALTYPE   WCHAR
        | gRegistryPathStore[256]={0};
67675|
67676| /*-----
        | -----*/
67677| void VdGetRegistrySettings ( IN PUNICODE_STRING
        | RegistryPath )
67678| {
67679|   PAGED_CODE();
67680|
67681|   | Reg_GetULONGKey(RegistryPath,L"AllowWrites",ALLOWWRITES_
        | DEF,&gAllowWrites);
67682| /*
67683|   | Reg_GetULONGKey(RegistryPath,L"NumberOfBuffers",NUMBEROF
        | BUFFERS_DEF,&gNumberOfBuffers);
67684|   | Reg_GetULONGKey(RegistryPath,L"ScanLuns",SCANLUNS_DEF,&g
        | ScanLuns);
67685|   | Reg_GetULONGKey(RegistryPath,L"BufferSize",BUFFERSIZE_DE
        | F,&gBufferSize);
67686|   | Reg_GetULONGKey(RegistryPath,L"WriteCacheMaxSize",WRITEC
        | ACHEMAXSIZE_DEF,&gWriteCacheMaxSize);
67687|   | Reg_GetULONGKey(RegistryPath,L"DoSeek",DOSEEK_DEF,&gDoSe
        | ek);
67688|   | Reg_GetULONGKey(RegistryPath,L"CacheFlags",CACHEFLAGS_DE
        | F,&gCacheFlags);
67689| */
67690| }
67691|
67692| /*-----
        | -----*/
67693| NTSTATUS
67694| InitVDisk(
67695|     IN PDRIVER_OBJECT DriverObject,
67696|     IN PUNICODE_STRING RegistryPath
67697| )

```

```

67698|
67699| /*++
67700|
67701| Routine Description:
67702|
67703|   This is the routine called by the system to
        | initialize the disk
67704|   performance driver. The driver object is set up and
        | then the
67705|   driver calls TdromInitialize to attach to the boot
        | devices.
67706|
67707|   IRQL = PASSIVELEVEL
67708| Arguments:
67709|
67710|   DriverObject - The disk performance driver object.
67711|
67712| Return Value:
67713|
67714|   NTSTATUS
67715|
67716| --*/
67717|
67718| {
67719|   NTSTATUS   ntStatus=0;
67720|   WCHAR Name[80]={0};
67721|   UNICODE_STRING NameUnicode={0};
67722|   NTSTATUS Status=0;
67723|   OBJECT_ATTRIBUTES ObjectAttributes={0};    // for
        | the directory object
67724|
67725|   NOT_REFERENCED(DriverObject);
67726|   PAGED_CODE();
67727|
67728|
        | RtlCopyMemory(gRegistryPathStore,RegistryPath->Buffer,Re
        | gistryPath->Length);
67729|   // NULL terminate, since counted unicode strings
        | are not necessarily null
67730|   // terminated
67731|   gRegistryPathStore[RegistryPath->Length / 2] = 0;
67732|
        | RtlInitUnicodeString(&gRegistryPath,gRegistryPathStore);
67733|
67734|   Debug(DEBUG_INIT,("RegistryPath =
        | %wZ\n",&gRegistryPath));
67735|   VdGetRegistrySettings ( &gRegistryPath );
67736|
67737|   // Create a permanent object directory for
        | partitions, then make it

```



```

67738| // temporary so that we can close it at any time
| and it will go away.
67739|
67740|
| swprintf(Name,L"\\Device\\PsmDevices_%%04x",PSM_LOW_COMPA
| TIBLE_VERSION);
67741|
67742| RtlInitUnicodeString( &NameUnicode, Name);
67743|
67744| InitializeObjectAttributes(
67745|             &ObjectAttributes,
67746|             &NameUnicode,
67747|             OBJ_PERMANENT,
67748|             NULL,
67749|             NULL );
67750|
67751| Status = ZwCreateDirectoryObject(
67752|
| &gVDiskRootDirHandle,
67753|
| DIRECTORY_ALL_ACCESS,
67754|             &ObjectAttributes
| );
67755|
67756| if ( NT_SUCCESS( Status ) ) {
67757|     ZwMakeTemporaryObject( gVDiskRootDirHandle );
67758| }
67759|
67760| //InitExit:
67761|
67762| Debug(DEBUG_INIT,("VDisk init returning %%08x to
| NT\\n",ntStatus));
67763| return(ntStatus);
67764|
67765| } // InitVDisk
67766|
67767|
67768| #if _WIN32_WINNT>=0x0500
67769| NTSTATUS
67770| PSMANAddDevice(
67771|     IN PDRIVER_OBJECT DriverObject,
67772|     IN PDEVICE_OBJECT PhysicalDeviceObject
67773| )
67774| /*++
67775| Routine Description:
67776|
67777| Creates and initializes a new filter device object
| FiDO for the
67778| corresponding PDO. Then it attaches the device
| object to the device

```

```

67779|    stack of the drivers for the device.
67780|
67781|    Note: You can NOT access the PDO as it has not yet
        | been assigned any resources or
67782|        started. Do device stuff in
        | IRP_MN_START_DEVICE
67783|
67784| Arguments:
67785|
67786|    DriverObject - Disk performance driver object.
67787|    PhysicalDeviceObject - Physical Device Object from
        | the underlying layered driver
67788|
67789| Return Value:
67790|
67791|    NTSTATUS
67792| --*/
67793| {
67794|     NTSTATUS          status;
67795|     PDEVICE_OBJECT    filterDeviceObject;
67796|     PFILTERED_EXTENSION deviceExtension;
67797|     PWMILIB_CONTEXT    wmlibContext;
67798|     PCHAR              buffer;
67799|     ULONG              buffersize;
67800|     ULONG              SizeOfDeviceExt;
67801|
67802|     PAGED_CODE();
67803|
67804|     //
67805|     // Create a filter device object for this device
        | (partition).
67806|     //
67807|
67808|     Debug(DEBUG_INIT,("PSManAddDevice: Driver %X Device
        | %X\n", DriverObject, PhysicalDeviceObject));
67809|
67810| #ifdef DEBUG_EXTENSION
67811|     SizeOfDeviceExt = sizeof(DEVICE_EXTENSION);
67812| #else
67813|     SizeOfDeviceExt = sizeof(FILTERED_EXTENSION);
67814| #endif
67815|     status = IoCreateDevice(DriverObject,
67816|                             SizeOfDeviceExt,
67817|                             NULL,
67818|                             FILE_DEVICE_DISK,
67819|                             0,
67820|                             FALSE,
67821|                             &filterDeviceObject);
67822|
67823|     if ( !NT_SUCCESS(status) ) {

```

```

67824|     Debug(DEBUG_INIT,("PSManAddDevice: Cannot
| create filterDeviceObject\n"));
67825| #ifdef DEBUG
67826|     | PSManBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,0x20,0
| ,0);
67827| #endif
67828|     return status;
67829| }
67830|
67831| filterDeviceObject->Flags |= DO_DIRECT_IO;
67832|
67833| #ifdef DEBUG_EXTENSION
67834|     | ((PDEVICE_EXTENSION)(filterDeviceObject->DeviceExtension
| ))->ObjectType = OBJECT_FILTEREDDISK;
67835|     | ((PDEVICE_EXTENSION)(filterDeviceObject->DeviceExtension
| ))->RealDeviceExtension =
| MemAllocatePoolWithTag(NonPagedPool,sizeof(FILTERED_EXTE
| NSION),DEVEXTTAG);
67836|     | RtlZeroMemory(((PDEVICE_EXTENSION)(filterDeviceObject->D
| eviceExtension))->RealDeviceExtension,sizeof(FILTERED_EX
| TENSION));
67837| #endif
67838|     deviceExtension = (PFILTERED_EXTENSION)
| GetDeviceExtension(filterDeviceObject);
67839|
67840|     RtlZeroMemory(deviceExtension,
| FILTERED_EXTENSION_SIZE);
67841|
67842|     deviceExtension->DeviceObject = filterDeviceObject;
67843|     deviceExtension->ObjectType =
| OBJECT_FILTEREDDISK;
67844|     deviceExtension->DiskNumber = -1;
67845|     // until we know better
67846|     deviceExtension->IsPhysical = FALSE;
67847|     deviceExtension->Physical.LastPartitionNumber = 0;
67848|     deviceExtension->DriverObject = DriverObject;
67849|
67850|     Debug(DEBUG_DEVCON,("AddDevice: directio=%d,
| Setting to false\n",deviceExtension->DoDirectIO));
67851|     deviceExtension->InLoadUnload = FALSE;
67852|     deviceExtension->DoDirectIO = FALSE;
67853|     deviceExtension->IsMounted = FALSE;
67854|     Debug(DEBUG_DEVCON,("PSManAddDevice: Just set
| IsMounted to FALSE for DevExt=%08x,
| DevObj=%08x\n",deviceExtension,filterDeviceObject));
67855|     Debug(DEBUG_DEVCON,("direct maps cleared\n"));

```

```

67856|
    | RtlZeroMemory(&deviceExtension->Cache,sizeof(tCacheInfo)
    | );
67857|
67858|     deviceExtension->Cache.HeaderFile.FileHandle =
67859|     deviceExtension->Cache.IndexFile.FileHandle =
67860|     deviceExtension->Cache.CacheFile.FileHandle =
    | INVALID_HANDLE_VALUE;
67861|
67862|     deviceExtension->Cache.HeaderFile.WaitHandle =
67863|     deviceExtension->Cache.IndexFile.WaitHandle =
67864|     deviceExtension->Cache.CacheFile.WaitHandle =
    | INVALID_HANDLE_VALUE;
67865|
67866|     ExInitializeResourceLite (
    | &(deviceExtension->Cache.DirectAccessResource) );
67867|     pmRegisterObject (
    | &(deviceExtension->Cache.DirectAccessResource),
    | "DirectAccessResource", pmRwLock );
67868|
67869|     InitializeListHead(&deviceExtension->Cache.SnapShotHead)
    | ;
67870|
67871|     deviceExtension->ChangeCount =          // for
    | removables.
67872|
    | deviceExtension->SignalRead =
67873|
    | deviceExtension->SignalWrite =
67874|
    | deviceExtension->PSMed      =
67875|
    | deviceExtension->NumberOfReadRequests =
67876|
    | deviceExtension->SectorsRead =
67877|
    | deviceExtension->NumberOfWriteRequests =
67878|
    | deviceExtension->SectorsWritten =
67879|
    | deviceExtension->OpenCount  = 0;        // number of
    | times it has been opened
67880|
67881|     deviceExtension->DeviceShutDown = 0;
67882|     // init linked list of snapshots.
67883|     InitializeListHead(&deviceExtension->SnapShots);
67884|
67885|     // initialize Read and write events
67886|     KeInitializeEvent ( &deviceExtension->ReadEvent,

```

```

    | SynchronizationEvent, FALSE );
67887| KeInitializeEvent ( &deviceExtension->WriteEvent,
    | SynchronizationEvent, FALSE );
67888|
67889| // Initialize spinlocks
67890|
67891| KeInitializeSpinLock (
    | &deviceExtension->StatisticsSpinLock );
67892|
    | pmRegisterObject(&deviceExtension->StatisticsSpinLock,"d
    | eviceExtension->StatisticsSpinLock",pmSpinLock);
67893|
67894| KeQuerySystemTime(&deviceExtension->LastIdleClock);
67895|
67896| //
67897| // Allocate per processor counters. NOTE: To save
    | some memory, it does
67898| // allocate memory beyond QueryTime. Remember to
    | expand size if there
67899| // is a need to use anything beyond this
67900| //
67901| deviceExtension->Processors = (ULONG)
    | *KeNumberProcessors;
67902| buffersize= PROCESSOR_COUNTERS_SIZE *
    | deviceExtension->Processors;
67903| buffer = (PCHAR)
    | MemAllocatePoolWithTag(NonPagedPool,
    | buffersize,TEMPTAG);
67904| if ( buffer != NULL ) {
67905|     RtlZeroMemory(buffer, buffersize);
67906|     deviceExtension->DiskCounters =
    | (PDISK_PERFORMANCE) buffer;
67907| } else {
67908|
    | LogError(filterDeviceObject,NULL,IO_ERR_INSUFFICIENT_RES
    | OURCES,STATUS_INSUFFICIENT_RESOURCES,NULL,0,NULL,0);
67909| }
67910|
67911| //
67912| // Attaches the device object to the highest device
    | object in the chain and
67913| // return the previously highest device object,
    | which is passed to
67914| // IoCallDriver when pass IRPs down the device
    | stack
67915| //
67916|
67917| deviceExtension->PhysicalDeviceObject =
    | PhysicalDeviceObject;
67918|

```

```

67919|   deviceExtension->TargetDeviceObject =
        | IoAttachDeviceToDeviceStack(filterDeviceObject,
        | PhysicalDeviceObject);
67920|
67921|   if ( deviceExtension->TargetDeviceObject == NULL )
        | {
67922|       IoDeleteDevice(filterDeviceObject);
67923|       Debug(DEBUG_INIT,("PSManAddDevice: Unable to
        | attach %X to target %X\n",filterDeviceObject,
        | PhysicalDeviceObject));
67924| #ifdef DEBUG
67925|
        | PSManBugCheck(SB_BUG_FILE_INIT,SB_BUG_INIT_FAILED,0x21,0
        | ,0);
67926| #endif
67927|       return STATUS_NO_SUCH_DEVICE;
67928|   }
67929|
67930|   //
67931|   // Save the filter device object in the device
        | extension
67932|   //
67933|   deviceExtension->DeviceObject = filterDeviceObject;
67934|
67935|   deviceExtension->PhysicalDiskIoNotifyRoutine =
        | NULL;
67936|   deviceExtension->PhysicalDeviceName.Buffer =
        | deviceExtension->PhysicalDeviceNameBuffer;
67937|
67938|
        | KeInitializeEvent(&deviceExtension->PagingPathCountEvent
        | , NotificationEvent, TRUE);
67939|
67940|   // per device resources that need to be freed
67941|   // this is freed no matter what when DELETE_DEVICE
        | is called...
67942|   // is this right??? FIXFIXFIX
67943|
        | ExInitializeResourceLite(&deviceExtension->DeviceExtReso
        | urce);
67944|
        | pmRegisterObject(&deviceExtension->DeviceExtResource,"de
        | viceExtension->DeviceExtResource",pmRwLock);
67945|
67946|   //
67947|   // Initialize WMI library context
67948|   //
67949|   wmlibContext = &deviceExtension->WmlibContext;
67950|   RtlZeroMemory(wmlibContext,
        | sizeof(WMILIB_CONTEXT));

```

```

67951|    wmlibContext->GuidCount = PSManGuidCount;
67952|    wmlibContext->GuidList = PSManGuidList;
67953|    wmlibContext->QueryWmiRegInfo =
        | PSManQueryWmiRegInfo;
67954|    wmlibContext->QueryWmiDataBlock =
        | PSManQueryWmiDataBlock;
67955|    wmlibContext->WmiFunctionControl =
        | PSManWmiFunctionControl;
67956|
67957|    //
67958|    // default to DO_POWER_PAGABLE
67959|    //
67960|
67961|    filterDeviceObject->Flags |= DO_POWER_PAGABLE;
67962|
67963|    //
67964|    // Clear the DO_DEVICE_INITIALIZING flag
67965|    //
67966|
67967|    filterDeviceObject->Flags &=
        | ~DO_DEVICE_INITIALIZING;
67968|
67969|    Debug(DEBUG_INIT,("PSManAddDevice: FilterObject %X
        | attached to target %X\n",filterDeviceObject,
        | PhysicalDeviceObject));
67970|    return STATUS_SUCCESS;
67971|
67972| }
67973| #endif
67974|
67975|
67976|
67977| File Listing: SBPSMAN.h
67978|
67979| #define SBPSMAN_WIN32_NAME L"\\DosDevices\\PSMan"
67980| #define SBPSMAN_DEVICE_NAME L"\\Device\\PSMan"
67981| #define _PROGRAMLONGNAME L"SnapBack Live"
67982| #define _PROGRAMSHORTNAME L"SnapBack"
67983|
67984| #define MAXDEVICES 32
67985|
67986| #define GLOBALTYPE
67987|
67988| #define MAX_PATH MAXIMUM_FILENAME_LENGTH
67989|
67990| #define NUMTHREADS_DEF 10
67991| #define NUMTHREADS_MIN 1
67992| #define NUMTHREADS_MAX 1000
67993|
67994| #define MAXTHREADS_DEF 32

```

```

67995| #define MAXTHREADS_MAX NUMTHREADS_MAX
67996|
67997| // number of microseconds to wait before creating
    | another thread when busy
67998| #define NEWTHREADDELAY_DEF 1000000
67999| #define NEWTHREADDELAY_MIN 1
68000| #define NEWTHREADDELAY_MAX 60000000
68001|
68002| // can be FILE_DELETE_ON_CLOSE and FILE_NO_COMPRESSION,
    | etc...
68003| #define CREATEOPTIONS_DEF (FILE_NO_COMPRESSION)
68004| #define OPENOPTIONS_DEF (FILE_NO_COMPRESSION |
    | FILE_WRITE_THROUGH | FILE_NO_INTERMEDIATE_BUFFERING)
68005|
68006| // timeout for when system is hung default is 30
    | seconds.
68007| #define HUNGSYSTEMTIMEOUT_DEF 30000000
68008| #define HUNGSYSTEMTIMEOUT_MIN 1
68009| #define HUNGSYSTEMTIMEOUT_MAX 2000000000
68010|
68011| // from tdrom.h
68012| #define NUMBEROFBUFFERS_DEF 0
68013| #define BUFFERSIZE_DEF (128*1024) /* 128k */
68014| #define ALLOWWRITES_DEF 0
68015| #define WRITECACHEMAXSIZE_DEF 2048 /* 1 MB (1048576
    | / 512) */
68016| #define DOSEEK_DEF 1
68017| #define SCANLUNS_DEF 1
68018| #define CACHEFLAGS_DEF 0
68019| #define MAKEPHYSICAL_DEF 0
68020|
68021|
68022|
68023| #define DEBUG_INFO 0x02000000
68024| #define DEBUG_ERROR 0x04000000
68025| #define DEBUG_PROCCALL 0x08000000
68026|
68027| #define DEBUG_INIT 0x00000001
68028| #define DEBUG_THREAD 0x00000002
68029| #define DEBUG_TREE 0x00000004
68030| #define DEBUG_FILE 0x00000008
68031| #define DEBUG_READ 0x00000010
68032| #define DEBUG_CACHE 0x00000020
68033| #define DEBUG_DEVCON 0x00000040
68034| #define DEBUG_REG 0x00000080
68035| #define DEBUG_DEVSUP 0x00000100
68036| #define DEBUG_SHUTDOWN 0x00000200
68037| #define DEBUG_CLEANUP 0x00000400
68038| #define DEBUG_MISC 0x00000800
68039| #define DEBUG_CREATE 0x00001000

```



```

68040| #define DEBUG_CLOSE      0x00002000
68041| #define DEBUG_FLUSH        0x00004000
68042| #define DEBUG_DCPSM        0x00008000
68043| #define DEBUG_WRITE         0x00010000
68044| #define DEBUG_VDISK         0x00020000
68045| #define DEBUG_MEMORY        0x00040000
68046| #define DEBUG_PASSTHRU      0x00080000
68047| #define DEBUG_PSMFILES      0x00100000
68048| #define DEBUG_PNP           0x00200000
68049| #define DEBUG_POWER         0x00400000
68050| #define DEBUG_WMI           0x00800000
68051| #define DEBUG_DICT          0x01000000
68052| #define DEBUG_SFILTER       0x02000000
68053|
68054| #define DEBUG_TRACE         0x80000000
68055|
68056|
68057| // Keep sorted so it is easy to see duplicates.
68058| // Each name looks backwards but it will be forwards in
    | little-endian.
68059|
68060| #define FREETAG             'EERF'
68061| #define SBMEM16TAG          '61SP'
68062| #define BUGCHECK_TAG        'cbSP'
68063| #define BITMAPTAG           'mbSP'
68064| #define BUFFTAG             'ubSP'
68065| #define DEBUG_ENTRY_TAG     'bdSP'
68066| #define DEVEXTTAG           'edSP'
68067| #define PSM_DISPATCH_TABLE_TAG 'tdSP'
68068| #define PSM_EVENT_ENTRY_TAG 'eeSP'
68069| #define PSM_EVENT_TAG       'teSP'
68070| #define EVENTTAG            'veSP'
68071| #define FILENAMETAG         'nfSP'
68072| #define PSM_FREE_SPACE_TAG   'sfSP'
68073| #define GETSTATSTAG         'sgSP'
68074| #define PSM_DICT_HEADER_TAG 'ehSP'
68075| #define IRPTAG              'riSP'
68076| #define PSM_INTERNAL_SNAPSHOT 'siSP'
68077| #define PSM_MASTER_SNAPSHOT  'amSP'
68078| #define MBRTAG              'bmSP'
68079| #define NODEBITTAG          'bnSP'
68080| #define NODEMEMTAG          'mnSP'
68081| #define NODETAG             'onSP'
68082| #define OPENTHREADTAG       'toSP'
68083| #define PAGEDSECTAG         'spSP'
68084| #define QTAG                 'tqSP'
68085| #define PSM_REVERT_BUFFER_TAG 'brSP'
68086| #define PSM_READONLY_FILE_TAG 'orSP'
68087| #define READREQUESTTAG      'rrSP'
68088| #define REGISTRYTAG         'trSP'

```

```

68089| #define PSMSECTORBITTAG    'bsSP'
68090| #define PSM_SECURITY_TAG    'esSP'
68091| #define PSM_DICT_SHARED_TAG    'hsSP'
68092| #define PSM_SKIP_FILE_TAG    'ksSP'
68093| #define PSM_SNAPSHOT_ENTRY    'ssSP'
68094| #define STRINGTAG            'tsSP'
68095| #define THREADTAG            'htSP'
68096| #define TEMPTAG              'mtSP'
68097| #define TreeTAG              'rtSP'
68098| #define USERTAG              'suSP'
68099| #define PSM_VDISK_BUFFER_TAG    'bvSP'
68100| #define VDISKWRITETAG        'wvSP'
68101| #define WRITEREQUESTTAG        'rwSP'
68102|
68103|
68104| #ifdef DEBUG
68105| #define TRACE(Code,Arg1,Arg2,Arg3,Arg4,Msg)
        | SbTrace2((Code),(Arg1),(Arg2),(Arg3),(Arg4),(Msg),__FILE
        | __,__LINE__)
68106| #else
68107| #define TRACE(Code,Arg1,Arg2,Arg3,Arg4,Msg)
68108| #endif
68109|
68110| // device.c
68111| #define TRACE_PASSTHRU            "PassThru" //
        | high, low, count, key
68112| #define TRACE_CREATE            "Create" //
        | Security, Options, Attributes, EaLength
68113| #define TRACE_READ              "Read" //
        | high, low, count, key
68114| #define TRACE_WRITE              "Write" //
        | high, low, count, key
68115| #define TRACE_READFORWRITE      "ReadForWrite"
        | // high, low, count, key
68116| #define TRACE_PASSTHRU_COMP      "PassThruComp"
        | // high, low, count, key
68117| #define TRACE_FLUSH              "Flush" // 0,
        | 0, 0, 0
68118| #define TRACE_IOCTL              "IoCtl" //
        | OutputBuffer Length, Input Buffer Length, Io Control
        | Code, Type 3 Input Buffer
68119| #define TRACE_SHUTDOWN          "Shutdown" //
        | 0, 0, 0, 0
68120| #define TRACE_CLEANUP            "Cleanup" //
        | 0, 0, 0, 0
68121| #define TRACE_CLOSE              "Close" // 0,
        | 0, 0, 0
68122|
68123| // Thread.c
68124| #define TRACE_THREADUPDATE        "ThreadUpdate"

```

```

| // Up, Threads Awake, Number of threads, 0
68125| #define TRACE_WRITEANDWAIT          "WriteAndWait"
| // high, low, count, 0
68126| #define TRACE_WRITEANDWAITCOMP
| "WriteAndWaitComp" // high, low, count, 0
68127| #define TRACE_LOOKINGFORDUPS        "LookingForDups"
| // 0, Sector, Count, 0
68128| #define TRACE_ADDTOTREE              "AddToTree" //
| 0, Sector, Count, Bit
68129| #define TRACE_THREADLOOP            "ThreadLoop"
| // 0, 0, 0, 0
68130| #define TRACE_GETWORK                "GetWork" //
| 0, 0, 0, 0
68131| #define TRACE_PROCESSECTOR           "ProcessSector"
| // 0, Sector, Count, 0
68132| #define TRACE_THREADCLEANUP          "ThreadCleanup"
| // 0, Sector, Count, 0
68133| #define TRACE_THREADEXIT              "ThreadExit"
| // 0, 0, 0, 0
68134| #define TRACE_MAKETHREAD             "MakeThread"
| // 0, 0, 0, 0
68135| #define TRACE_WAITWRITEFROMNT
| "WaitForWriteFromNT" // 0, 0, 0, 0
68136| #define TRACE_COMPLETENEXTWRITEONQUEUE
| "CompleteNextWrite" // 0, 0, 0, 0
68137| #define TRACE_COMPLETEWRITESONQUEUE
| "CompleteAllWrites" // 0, 0, 0, 0
68138| #define TRACE_ALLOCATERESOURCES
| "AllocateResources" // 0, 0, Byte Count, 0
68139| #define TRACE_WORKERGET               "WorkerGet" //
| 0, 0, 0, 0
68140| #define TRACE_SETTINGUP              "SettingUp" //
| 0, Sector, Count, 0
68141| #define TRACE_SENDINGREADFORWRITE
| "SendingReadForWrite" // 0, Sector, Count, 0
68142| #define TRACE_READFORWRITE_COMP
| "ReadForWriteComp" // 0, Sector, Count, 0
68143| #define TRACE_SENDINGORIGWRITE
| "SendingOrigWrite" // high, low, count, key
68144| #define TRACE_SENDINGORIGWRITECOMP
| "SendingOrigWriteComp" // high, low, count, key
68145| #define TRACE_READANDWAIT            "ReadAndWait"
| // high, low, count, 0
68146|
68147|
68148|
68149| // time manipulation macros.
68150| // NT keeps track of time in 100 Nanosecond increments.
68151| // this means the Smallest nano second is a multiple of
| 100, so

```

```

68152| // only use microseconds and above, unless you need
    | it...
68153| #define ABSOLUTE(wait)    (wait)
68154| #define RELATIVE(wait)    (-wait)
68155|
68156| #define NANOSECONDS(nanos) (((signed
    | __int64)(nanos)) / 100L)
68157| #define MICROSECONDS(micros) (((signed
    | __int64)(micros)) * NANOSECONDS(1000L))
68158| #define MILLISECONDS(milli) (((signed
    | __int64)(milli)) * MICROSECONDS(1000L))
68159| #define SECONDS(seconds) (((signed
    | __int64)(seconds)) * MILLISECONDS(1000L))
68160|
68161| /*lint -emacro(740,Debug)*/
68162| #ifdef DEBUG
68163|     void BugCheckCallBack( PVOID Buffer, ULONG Length );
68164|     void DebugPrintSave(char *fmt, ...);
68165| // #define Debug(level,_x_) if ((DebugLevel &
    | (level)) == (level)) DbgPrint _x_
68166| #define Debug(level,_x_) if ((DebugLevel & (level))
    | == (level)) DebugPrintSave _x_
68167| #else
68168| #define Debug(level,_x_)
68169| #endif
68170|
68171| #define _STATE_UNKNOWN          0
68172| #define _STATE_WAITING_FOR_WORK    1
68173| #define _STATE_WAITING_FOR_THREAD  2
68174| #define _STATE_WAITING_FOR_WRITE   3
68175| #define _STATE_WAITING_FOR_EVENT   4
68176| #define _STATE_WAITING_FOR_TREE_READ 5
68177| #define _STATE_WAITING_FOR_TREE_WRITE 6
68178| #define _STATE_WAITING_FOR_CACHE_BIT 7
68179| #define _STATE_WAITING_FOR_MEMORY   8
68180| #define _STATE_WORKING             50
68181| #define _STATE_CLEANUP             51
68182| #define _STATE_DEINIT              254
68183| #define _STATE_INIT                255
68184|
68185| typedef struct sMY_THREAD {
68186|     PVOID ThreadObject;
68187| #ifdef SYNC
68188|     PVOID FileObject;
68189| #endif
68190| #ifdef DEBUG
68191|     ULONG State;
68192|     ULARGE_INTEGER Sector;
68193|     ULONG Count;
68194|     ULARGE_INTEGER Current;

```

```

68195| PDEVICE_OBJECT DeviceObject;
68196| PIRP Irp;
68197| #endif
68198| } tMY_THREAD, *PMY_THREAD;
68199|
68200| #ifdef DEBUG
68201| extern GLOBALTYPE ULONG DebugLevel;
68202| extern GLOBALTYPE ULONG DebugPrints;
68203|
68204| extern GLOBALTYPE ULONG FIRST_DATA_SEG_ADDR;
68205| extern GLOBALTYPE ULONG DATA_SEG_ADDR1;
68206| extern GLOBALTYPE ULONG DATA_SEG_ADDR2;
68207| extern GLOBALTYPE ULONG DATA_SEG_ADDR3;
68208| extern GLOBALTYPE ULONG DATA_SEG_ADDR4;
68209| extern GLOBALTYPE ULONG DATA_SEG_ADDR5;
68210| extern GLOBALTYPE ULONG DATA_SEG_ADDR6;
68211| extern GLOBALTYPE ULONG DATA_SEG_ADDR7;
68212| extern GLOBALTYPE ULONG DATA_SEG_ADDR8;
68213| extern GLOBALTYPE ULONG DATA_SEG_ADDR9;
68214| extern GLOBALTYPE ULONG DATA_SEG_ADDR_A;
68215| extern GLOBALTYPE ULONG DATA_SEG_ADDR_B;
68216| extern GLOBALTYPE ULONG DATA_SEG_ADDR_C;
68217| extern GLOBALTYPE ULONG DATA_SEG_ADDR_D;
68218| extern GLOBALTYPE ULONG DATA_SEG_ADDR_E;
68219| extern GLOBALTYPE ULONG DATA_SEG_ADDR_F;
68220| extern GLOBALTYPE ULONG DATA_SEG_ADDR_G;
68221| extern GLOBALTYPE ULONG DATA_SEG_ADDR_H;
68222| extern GLOBALTYPE ULONG DATA_SEG_ADDR_I;
68223| extern GLOBALTYPE ULONG DATA_SEG_ADDR_J;
68224| extern GLOBALTYPE ULONG DATA_SEG_ADDR_K;
68225| extern GLOBALTYPE ULONG DATA_SEG_ADDR_L;
68226| extern GLOBALTYPE ULONG DATA_SEG_ADDR_M;
68227| extern GLOBALTYPE ULONG DATA_SEG_ADDR_N;
68228| extern GLOBALTYPE ULONG DATA_SEG_ADDR_O;
68229| extern GLOBALTYPE ULONG DATA_SEG_ADDR_P;
68230| extern GLOBALTYPE ULONG END_DATA_SEG_ADDR;
68231|
68232|
68233| #endif
68234|
68235| extern GLOBALTYPE WCHAR
| gSnapshotDirName[200];
68236| extern GLOBALTYPE PDRIVER_OBJECT
| PSMANDriverObject; // our driver object
68237| extern GLOBALTYPE PDEVICE_OBJECT PSMANObject;
| // our object that user opens
68238| extern GLOBALTYPE BOOLEAN PsmActive;
| // is psm enabled?
68239| extern GLOBALTYPE BOOLEAN PSMANPSMInitd;
| // Is psm ready?

```

```

68240| extern GLOBALTYPE  KSEMAPHORE      ThreadSemaphore;
    | // limits threads for "writing to cache file" that can
    | run at once
68241| extern GLOBALTYPE  KSEMAPHORE      WriteSemaphore;
    | // Number of writes pending for "write dispatch" to
    | threads
68242| extern GLOBALTYPE  KSEMAPHORE
    | WriteAfterReadSemaphore;    // Number of writes
    | pending for dispatch to lower driver
68243| extern GLOBALTYPE  KEVENT
    | WorkerThreadEvent;  // Set when a thread is available
    | to work
68244| extern GLOBALTYPE  KEVENT          Thread0Inited;
    | // used to sync threads
68245| extern GLOBALTYPE  KEVENT
    | PSMANExitingEvent;  // Set when time for psm to be
    | disabled (not actual unload of psm)
68246| extern GLOBALTYPE  ULONG          NumberOfThreads;
    | // Number of Threads for psm
68247| extern GLOBALTYPE  ULONG
    | GlobalThreadCount;  // Number of threads running
68248| extern GLOBALTYPE  PMY_THREAD      ThreadObjects;
    | // List if each threads data
68249| extern GLOBALTYPE  tMY_THREAD
    | WriteThreadObject;  // writer thread data
68250| extern GLOBALTYPE  LIST_ENTRY
    | ThreadsWorkToDoQueue; // Queue for threads
68251| extern GLOBALTYPE  KSPIN_LOCK
    | ThreadsWorkToDoSpinLock; // spinlock
68252| extern GLOBALTYPE  LIST_ENTRY
    | WriteAfterReadQueue;  // Queue for writes to be sent
    | after a read has occurred
68253| extern GLOBALTYPE  KSPIN_LOCK
    | WriteAfterReadSpinLock; // spinlock
68254| extern GLOBALTYPE  LIST_ENTRY      ProcessingQueue;
    | // Processing queue (from nt system)
68255| extern GLOBALTYPE  LIST_ENTRY      WriteQueue;
    | // Write queue (from nt system)
68256| extern GLOBALTYPE  KSPIN_LOCK      WriteSpinLock;
    | // spin lock for it
68257| extern GLOBALTYPE  ULONG          DoPagingFile;
    | // Set if we need to do the paging file
68258| extern volatile GLOBALTYPE  ULONG
    | OutstandingRequests;  // Number of IOs outstanding
    | to all volumes
68259| extern GLOBALTYPE  ULONG          PSMANPSMFlags;
    | // see PSM_FLAG_*
68260| extern GLOBALTYPE  FAST_MUTEX
    | WorkerThreadMutex;  // Mutex for threads
68261| extern GLOBALTYPE  FAST_MUTEX

```

```

    | CacheThresholdMutex;    // Mutex for threshold
    | handling
68262| extern GLOBALTYPE  FAST_MUTEX      PSMUserMutex;
    | // Mutex to keep track of psm users
68263| extern GLOBALTYPE  KSEMAPHORE
    | PSMOpenCloseSemaphore; // Keep open/close out of each
    | others hair
68264| #ifdef DEBUG
68265| extern GLOBALTYPE  KSEMAPHORE
    | PSM_DebugSemaphore; // Semaphore to write to log
    | file
68266| extern GLOBALTYPE  ULONG           gDebugToLog;
    | // whether to log debug to log file or not
68267| extern GLOBALTYPE  LIST_ENTRY      PSM_DebugQueue;
    | // Queue for threads
68268| extern GLOBALTYPE  KSPIN_LOCK
    | PSM_DebugSpinLock;    // spinlock
68269| #endif
68270| extern GLOBALTYPE  ULONG           gLogErrors;
68271| extern GLOBALTYPE  ULONG           gFailFreed;
68272| extern GLOBALTYPE  ULONG           gLogOpenClose;
68273| extern GLOBALTYPE  KSEMAPHORE
    | PSMVDiskSemaphore;    // Semaphore for read/write
    | vdisk volumes
68274| #ifdef DEBUG_SNAPSHOTS
68275| extern GLOBALTYPE  KSEMAPHORE
    | PSMSnapShotSemaphore; // Semaphore for read/write
    | snapshots
68276| #endif
68277| extern GLOBALTYPE  NTSTATUS      LastErrorStatus;
    | // error to report to PSM users
68278| extern GLOBALTYPE  FAST_MUTEX
    | PSMManMemoryMutex;    // mutex for memory routines
68279| extern GLOBALTYPE  ULONG
    | MaxWriteQueueDepth;   // stats for writes
68280| extern GLOBALTYPE  ULONG      WriteQueueDepth;
    | // stats
68281| extern volatile GLOBALTYPE  ULONG
    | ThreadsAwake;        // Number of threads
    | "actively" processing requests
68282| extern GLOBALTYPE  ULONG
    | NewThreadStartDelay;  // Number of milliseconds to
    | start a new thread when all others are busy
68283| extern GLOBALTYPE  ULONG      MaxThreads;
    | // Max number of threads to create
68284| extern GLOBALTYPE  ULONG
    | PSMManCreateOptions;  // Options for "creating" the
    | cache file
68285| extern GLOBALTYPE  ULONG
    | PSMManOpenOptions;    // Options for "opening" the

```

```

| cache file
68286| extern GLOBALTYPE  ULONG
| PSMANFillOnWrite;    // Actually write to the cache
| file?
68287| extern GLOBALTYPE  ULONG
| gHungSystemTimeOut;  // Number of milliseconds to
| wait to see if the system is deadlocked
68288| extern GLOBALTYPE  ULONG
| gVDiskIOHandling;    // when set any reads/writes
| to vdisk are completed successfully without IO
68289|
68290| // from tdrom.h
68291| extern GLOBALTYPE  KEVENT
| VDiskExitingEvent;   // Vdisks told to disappear
68292|
68293| extern GLOBALTYPE  LIST_ENTRY      ReadVDiskQueue;
| // Queue for reads to virtual volume
68294| extern GLOBALTYPE  KSPIN_LOCK
| ReadVDiskSpinLock;   // keep in sync
68295| extern GLOBALTYPE  KSEMAPHORE
| ReadVDiskSemaphore;  //
68296|
68297| extern GLOBALTYPE  LIST_ENTRY      WriteVDiskQueue;
| // Queue for writes to virtual volume
68298| extern GLOBALTYPE  KSPIN_LOCK
| WriteVDiskSpinLock;  // keep in sync
68299| extern GLOBALTYPE  KSEMAPHORE
| WriteVDiskSemaphore;
68300|
68301| extern GLOBALTYPE  ULONG
| VDiskNumberOfThreads; // Number of threads running
| (2, reader and writer)
68302| extern GLOBALTYPE  FAST_MUTEX
| VDiskThreadMutex;    // Keep in sync
68303|
68304| //extern GLOBALTYPE  ULONG
| gNumberOfBuffers;    // not used anymore
68305| //extern GLOBALTYPE  ULONG      gBufferSize;
| // not used anymore
68306|
68307| extern GLOBALTYPE  ULONG      gAllowWrites;
| // allow writes to virtual volumes? we now allow
| writes to all volumes, but
68308|
| // left here to show how to make volumes read only
68309| //extern GLOBALTYPE  ULONG
| gWriteCacheMaxSize;  // not used anymore
68310| //extern GLOBALTYPE  ULONG      gScanLuns;
| // not used
68311|

```



```

68312| extern GLOBALTYPE  UNICODE_STRING   gRegistryPath;
      | // our entry in the registry
68313|
68314| //extern GLOBALTYPE  struct sCache    VDiskCache;
68315| extern GLOBALTYPE  HANDLE
      | gVDiskRootDirHandle;   // our "\\Devices\\PsmDevices\\"
      | directory handle
68316|
68317| //extern GLOBALTYPE  ULONG            gDoSeek;
      | // Not used
68318| //extern GLOBALTYPE  ULONG            gCacheFlags;
      | // Not used
68319| // end from tdrom.h
68320|
68321| //
68322| // Device Extension
68323| //
68324|
68325| #define OBJECT_INTERNAL    0
68326| #define OBJECT_FILTEREDDISK 1
68327| #define OBJECT_VIRTUALDISK 2
68328| #define OBJECT_FS_OBJECT   3
68329| #define OBJECT_FS_FILTER   4
68330|
68331| // generic structure that is common to all extensions.
68332| typedef struct _DEVICE_EXTENSION {
68333|
68334|   PDEVICE_OBJECT DeviceObject;      // Back
      | pointer to device object
68335|   PDRIVER_OBJECT DriverObject;      // The
      | driver object for use on repartitioning.
68336|   ULONG           ObjectType;
68337|
68338| #ifdef DEBUG_EXTENSION
68339|   PVOID           RealDeviceExtension;
68340| #endif
68341| } DEVICE_EXTENSION, *PDEVICE_EXTENSION;
68342|
68343| typedef struct _PHYSICAL_DEVICE {
68344|   ULONG           Signature;
68345|   ULONG           LastPartitionNumber;
68346| } tPHYSICAL_DEVICE, *pPHYSICAL_DEVICE;
68347|
68348| typedef struct _MY_DISK_EXTENTS {
68349|   // from DISK_EXTENTS in ntddvol.h
68350|   ULONG           DiskNumber;
68351|   LARGE_INTEGER   StartingOffset;
68352|   LARGE_INTEGER   ExtentLength;
68353|
68354|   LIST_ENTRY      ListEntry;

```

```

68355| } MY_DISK_EXTENTS, *pMY_DISK_EXTENTS;
68356|
68357| typedef struct _LOGICAL_DEVICE {
68358| #if _WIN32_WINNT >= 0x0500
68359|     LIST_ENTRY    DiskExtents;
68360| #else
68361|     ULONG    Placeholder;
68362| #endif
68363| } tLOGICAL_DEVICE, *pLOGICAL_DEVICE;
68364|
68365| #define PSMAN_MAXSTR    64
68366|
68367| //
68368| // Disk notification or callout
68369| //
68370|
68371| typedef
68372| VOID
68373| (*PPHYSICAL_DISK_IO_NOTIFY_ROUTINE)(
68374|     IN ULONG DiskNumber,
68375|     IN PIRP Irp,
68376|     IN PDISK_PERFORMANCE PerfCounters
68377| );
68378|
68379| typedef struct PersistentDictionary::sHeader *pHeader;
68380| typedef struct PersistentDictionary::sInternalSnapShot
68381|     | *pInternalSnapShot;
68382|
68382| #define CACHE_ACTION_NOTHING    0
68383| #define CACHE_ACTION_DENY_WRITES    1
68384| #define CACHE_ACTION_BSOD    2
68385| #define CACHE_ACTION_DELETE_ALWAYS_KEEPS    3
68386|
68387| struct RETRIEVAL_POINTERS_BUFFER;
68388|
68389| typedef struct _PSM_FILE_INFO {
68390|     PFILE_OBJECT    FileObject;
68391|     HANDLE    FileHandle;
68392|     PVOID    WaitObject;
68393|     HANDLE    WaitHandle;
68394|     DirectAccessFile * Direct;
68395|     WCHAR    FileName[256];
68396|     WCHAR    Location[256];
68397| } tPsmFileInfo, *pPsmFileInfo;
68398|
68399| typedef struct _CACHE_INFO {
68400|     tPsmFileInfo    HeaderFile;
68401|     tPsmFileInfo    IndexFile;
68402|     tPsmFileInfo    CacheFile;
68403|     ERESOURCE    DirectAccessResource; // for

```

| doing reader-writer locks on usage of DirectAccessFile

| objects

```
68404|
68405|  FAST_MUTEX  PSMANBitMapMutex;
68406|  ULONG CurrentCacheFileSize;
68407|  ULONG PSMANBitHint;
68408|  PRTL_BITMAP PSMANBitMapBuffer;
68409|  ULONG PSMANCacheCanGrow;
68410|  pHeader Header;
68411|  ULONG HeaderReferenceCount;
68412|  ULONG ReferenceCount;
68413|  LIST_ENTRY SnapshotHead;
68414|  ULONG PSMANBitMapSize;
68415|  ULONG PSMANBitMapMaxSize;
68416|  ULONGLONG MostRecentSnapshotTime;
68417|  pInternalSnapshot MostRecentSnapshot;
68418|
68419|
68420|  // config stuff from registry
68421|  ULONG CacheWarningThresholdPercent;
68422|  ULONG CacheFullThresholdPercent;
68423|  ULONG CacheWarningInterval;
68424|  ULONG CacheFullActionPercent;
68425|  ULONG CacheFullAction;
68426|  ULONG InitialSize;
68427|  ULONG MaxSize;
68428| } tCacheInfo,*pCacheInfo;
68429|
68430|
68431| typedef struct _FILTERED_EXTENSION {
68432|  // common header
68433|  PDEVICE_OBJECT DeviceObject;      // Back
    | pointer to device object
68434|  PDRTIVER_OBJECT DriverObject;      // The
    | driver object for use on repartitioning.
68435|  ULONG      ObjectType;
68436|
68437|  // extension specific
68438|
68439|  // global to all filtered devices
68440|  PDEVICE_OBJECT TargetDeviceObject; // Target
    | Device Object
68441| #if _WIN32_WINNT < 0x0500
68442|  PDEVICE_OBJECT PhysicalDevice;     // Target
    | Device Object
68443| #endif
68444|  ULONG      IsPhysical:1;           // Device
    | is physical device
68445|  ULONG      NotActive:1;
68446|  ULONG      InLoadUnload:1;        // set when
```

```

| [Load|Unload]SnapShotsForVolume is running
68447|  ULONG      DoDirectIO:1;      // when
| set, we will do direct io to the files
68448|  ULONG      IsMounted:1;      // whether
| volume is mounted or not
68449|  ULONG      IsReverting:1;      // whether
| this volume is currently being reverted
68450|  ULONG      OpenCloseAcquired:1; // whether
| the open close resource is acquired, this is to get
| around a deadlock
68451|                                     // when
| opening my files, and the volume isnt mounted, so i
| then get a mount
68452|  ULONG      Dismounting:1;
68453|
68454|  ULONG      DiskNumber; // only valid if
| IsPhysical is set
68455|
68456|  UNICODE_STRING  UniName;
68457|  WCHAR          Name[256];
68458|
68459|  KSPIN_LOCK      StatisticsSpinLock;
68460|  ULONG           NumberOfReadRequests;
68461|  ULONG           SectorsRead;
68462|  ULONG           NumberOfWriteRequests;
68463|  ULONG           SectorsWritten;
68464|  ULONG           ChangeCount;      // used for
| removable devices
68465|
68466|  ULONG           CacheWrites;      // number of
| writes sent to cache file for this device
68467|
68468|  ULONG           SignalRead;
68469|  ULONG           SignalWrite;
68470|  ULONG           PSMed;
68471|  ULONG           OpenCount;      // number of
| times it has been opened
68472|
68473|  KEVENT          ReadEvent;      // a read occured
68474|  KEVENT          WriteEvent;     // a write occured
68475|
68476|  PARTITION_INFORMATION Pi;      // Partition
| info
68477|  LARGE_INTEGER   Cylinders;
68478|  MEDIA_TYPE      MediaType;
68479|  ULONG           TracksPerCylinder;
68480|  ULONG           SectorsPerTrack;
68481|  ULONG           BytesPerSector;
68482|
68483|  union {

```

```

68484|     tPHYSICAL_DEVICE Physical;
68485|     tLOGICAL_DEVICE Logical;
68486| };
68487|
68488|     ERESOURCE DeviceExtResource; // used for
    | misc changes to pointers in this structure.
68489|                                     // that can
    | change OUTSIDE of open and close
68490|
68491|     tCacheInfo Cache;
68492|     ULONG FileSystem; // file system
    | that will attach to us. see FILE_SYSTEM_*
68493|     LARGE_INTEGER LastMountTime;
68494|     // Valid only when PSM enabled
68495|
68496|     LIST_ENTRY SnapShots; // linked list of
    | snapshots. Newer times at head of list
68497|
68498| #if _WIN32_WINNT >= 0x0500
68499|     //
68500|     // Physical device object
68501|     //
68502|     PDEVICE_OBJECT PhysicalDeviceObject;
68503|
68504|     //
68505|     // Use to keep track of Volume info from ntddvol.h
68506|     //
68507|
68508|     WCHAR StorageManagerName[8];
68509|
68510|     //
68511|     // Disk performance counters
68512|     // and locals used to compute counters
68513|     //
68514|
68515|     ULONG Processors;
68516|     PDISK_PERFORMANCE DiskCounters; // per processor
    | counters
68517|     LARGE_INTEGER LastIdleClock;
68518|     LONG QueueDepth;
68519|     LONG CountersEnabled;
68520|
68521|     //
68522|     // must synchronize paging path notifications
68523|     //
68524|     KEVENT PagingPathCountEvent;
68525|     ULONG PagingPathCount;
68526|
68527|     //
68528|     // Physical Device name or WMI Instance Name

```

```

68529|  //
68530|
68531|  UNICODE_STRING PhysicalDeviceName;
68532|  WCHAR PhysicalDeviceNameBuffer[PSMAN_MAXSTR];
68533|
68534|  //
68535|  // Notification routine for tracing
68536|  //
68537|  PPHYSICAL_DISK_IO_NOTIFY_ROUTINE
    | PhysicalDiskIoNotifyRoutine;
68538|
68539|  //
68540|  // Private context for using WmiLib
68541|  //
68542|  WMILIB_CONTEXT WmilibContext;
68543|
68544|  // unique volume id for this volume, based on the
    | volume guid
68545|  ULONG   Volumeld;
68546|  WCHAR   VolumeGuid [40];  // string of the form
    | 'c8e56d0c-93c5-11d4-9910-806d6172696f'
68547| #define LENGTH_OF_UNIQUE
    | (((16*sizeof(WCHAR))*2)+(sizeof(WCHAR)*2))
68548|  WCHAR   UniqueId[LENGTH_OF_UNIQUE]; // string in
    | the form of '123456781234567812345678'
68549| #endif
68550|  ULONG DeviceShutDown;
68551|
68552|  ULONG NextIndexSequenceNumber;
68553|
68554| } FILTERED_EXTENSION, *PFILTERED_EXTENSION;
68555|
68556| #define FILTERED_EXTENSION_SIZE
    | sizeof(FILTERED_EXTENSION)
68557|
68558| typedef struct _FS_FILTER_EXTENSION {
68559|  // common header
68560|  PDEVICE_OBJECT DeviceObject;      // Back
    | pointer to device object
68561|  PDRIVER_OBJECT DriverObject;      // The
    | driver object
68562|  ULONG           ObjectType;
68563|
68564|  // extension specific
68565|  PDEVICE_OBJECT TargetDeviceObject;
68566|  ULONG Attached:1;                // If this
    | device is attached to a lower device
68567|  ULONG Virtual:1;                 // whether
    | the storage class object is a filtered disk or a
    | virtual disk

```

```

68568|  ULONG FileSystem:1;           // whether
      | this is a file system filter, or volume filter
68569|  ULONG IsRaw:1;                 // whether
      | this file system is the raw file system
68570|  ULONG IsNtfs:1;
68571|  ULONG IsFat:1;
68572|  PDEVICE_OBJECT PSMStorageObject; // our
      | storage class object that this file system filter is
      | filtering
68573| } FS_FILTER_EXTENSION, *PFS_FILTER_EXTENSION;
68574|
68575| #define FS_FILTER_EXTENSION_SIZE
      | sizeof(FS_FILTER_EXTENSION)
68576|
68577| typedef struct _FS_OBJECT_EXTENSION {
68578|     // common header
68579|     PDEVICE_OBJECT DeviceObject;      // Back
      | pointer to device object
68580|     PDRIVER_OBJECT DriverObject;      // The
      | driver object
68581|     ULONG           ObjectType;
68582|
68583|     // extension specific
68584| } FS_OBJECT_EXTENSION, *PFS_OBJECT_EXTENSION;
68585|
68586| #define FS_OBJECT_EXTENSION_SIZE
      | sizeof(FS_OBJECT_EXTENSION)
68587|
68588|
68589| #define PROCESSOR_COUNTERS_SIZE
      | FIELD_OFFSET(DISK_PERFORMANCE, QueryTime)
68590|
68591| typedef struct _OT_USER_ {
68592|     struct _OT_USER_ *Next;
68593|     PEPROCESS        ProcessID;
68594|     PETHREAD         ThreadID;
68595|     ULONG             Open; // if psm was opened by
      | this user
68596|     PFILE_OBJECT      FileObject; // file object
      | associated with the create request.
68597|     PKEVENT           ErrorEvent;
68598|     PKEVENT           AbortEvent;
68599|     ULONG             NumOpenSnapShots;
68600|     LIST_ENTRY        SnapShots;
68601|     ULONG             Persistent:1;
68602|     ULONG             SaveTempOnExit:1;
68603| } tOT_USER, *pOT_USER;
68604|
68605| // global memory
68606| // dont change the position of DeviceObject as this

```

```

| structure is
68607| // used as a DEVICE_EXTENSION in some places (ie
| unload)
68608|
68609| typedef struct _SBPSMAN_EXTENSION {
68610|     // common header
68611|     PDEVICE_OBJECT DeviceObject;        // 0 Back
| pointer to device object
68612|     PDIRECTOR_OBJECT DriverObject;      // 4 The
| driver object for use on repartitioning.
68613|     ULONG           ObjectType;         // 8
68614|
68615|     // extension specific
68616|     KSPIN_LOCK      WaitQueueSpinLock; // c
68617|     LIST_ENTRY       WaitQueue;        // 10
68618|
68619|     ULONG           LargestBPS;        // 18
68620|     ERESOURCE        DeviceResource;    // 1c used
| to sync access for psm
68621|     ERESOURCE        SnapShotResource;  // 54 used
| to sync access to snapshots
68622|
68623|     pOT_USER         PSMUsers;         // 8c list
| of threads who have psm open.
68624|     ULONG           NumActive;         // 90
| number of active opens
68625|     ULONG           ShutDownCalled;
68626|
68627| } SBPSMAN_EXTENSION, *PSBPSMAN_EXTENSION;
68628|
68629| #define SBPSMAN_EXTENSION_SIZE
| sizeof(SBPSMAN_EXTENSION)
68630|
68631| typedef struct _VDISK_EXTENSION {
68632|     // common header
68633|     PDEVICE_OBJECT DeviceObject;        // Back
| pointer to device object
68634|     PDIRECTOR_OBJECT DriverObject;      // The
| driver object for use on repartitioning.
68635|     ULONG           ObjectType;
68636|
68637|     // extension specific
68638|     PDEVICE_OBJECT PSMDevice;          // device
| being PSMD
68639|     ULONG           IsPhysical:1;       // Device
| is physical device
68640|     ULONG           MountDisabled:1;    // we set
| this bit to refuse mounts in certain situations
68641|
68642|     KSPIN_LOCK      StatisticsSpinLock;

```



```

68643|  ULONG      NumberOfReadRequests;
68644|  ULONG      SectorsRead;
68645|  ULONG      NumberOfWriteRequests;
68646|  ULONG      SectorsWritten;
68647|
68648|  tkSnapshotEntry *SnapShot; // pointer to snapshot
    | this volume is suppose to be of.
68649|  tkSnapshotMaster *MasterSnapShot; // Master
    | pointer for quick reference
68650|
68651|  ULONG      SerialNumber;
68652|
68653|  PARTITION_INFORMATION Pi;          // Partition
    | info
68654|  LARGE_INTEGER Cylinders;
68655|  ULONG      Heads;
68656|  ULONG      SPT;
68657|  ULONG      BPS;
68658|
68659|  ULONG      Instance;
68660|  WCHAR      Name[256];
68661|  WCHAR      VolumeGuid[100];
68662|
68663|  BOOLEAN     DriveNotReady;
68664|  BOOLEAN     PartitionActive;
68665|  ULONG      DiskChangeCount;
68666|  ULONG      LockCount;
68667|  ULONG      KeepWriteInMemory;
68668|  BOOLEAN     OriginalWriteProtected;
68669|
68670|  // how much went to cache because of "write" to
    | virtual drive
68671|  ULONG      CacheWrites;
68672|  ULONG      Cluster0Offset;
68673|
68674|  ULONG      DeviceShutDown;
68675|
68676| } VDISK_EXTENSION, *PVDISK_EXTENSION;
68677|
68678| #define VDISK_EXTENSION_SIZE sizeof(VDISK_EXTENSION)
68679|
68680|
68681| typedef struct sWriteRequest {
68682|  PDEVICE_OBJECT DeviceObject;
68683|  PIRP           Irp;
68684|  LIST_ENTRY     ListEntry;
68685|  LIST_ENTRY     ProcessingEntry;
68686|  ULARGE_INTEGER RoundedSector;    // starting
    | sector to pre-read; always on a granule boundary
68687|  ULONG      RoundedCount;    // number

```

```

    | of sectors to pre-read; always an integer number of
    | granules
68688|  LARGE_INTEGER  RoundedSectorInBytes;  //
    | RoundedSector multiplied by bytes/sector
68689|  LARGE_INTEGER  RoundedCountInBytes;  //
    | RoundedCount multiplied by bytes/sector
68690|  ULARGE_INTEGER RealSector;           // original
    | write request's starting sector
68691|  ULONG          RealCount;             // original
    | write request's sector count
68692|  PVOID          Buffer;
68693|  LARGE_INTEGER  ByteOffset;
68694|  ULONG          ByteLength;
68695| } tWriteRequest;
68696|
68697| #define tReadRequest tWriteRequest
68698|
68699| typedef struct sOpenPsmThread {
68700|  PIRP  Irp;
68701|  pOT_USER User;
68702|  PVOID  OTI;
68703|  ULONG  OTISize;
68704|  ULONG  OTOSize;
68705| } tOpenPsmThread, *pOpenPsmThread;
68706|
68707| typedef struct sDebugLogEntry {
68708|  LIST_ENTRY ListEntry;
68709|  char LogEntry[255];
68710| } tDebugLogEntry;
68711|
68712|
68713| //
68714| // Function declarations
68715| //
68716|
68717|
68718| extern "C" NTSTATUS
68719| DriverEntry(
68720|  IN PDRIVER_OBJECT DriverObject,
68721|  IN PUNICODE_STRING RegistryPath
68722| );
68723|
68724| VOID
68725| PSMANInitialize(
68726|  IN PDRIVER_OBJECT DriverObject,
68727|  IN PVOID NextDisk,
68728|  IN ULONG Count
68729| );
68730|
68731| NTSTATUS

```

```

68732| InitVDisk(
68733|     IN PDRIVER_OBJECT DriverObject,
68734|     IN PUNICODE_STRING RegistryPath
68735| );
68736| NTSTATUS PSMANMakePartitionObjects(
68737|     PDEVICE_OBJECT Partition0,
68738|     BOOLEAN BootTime
68739| );
68740|
68741|
68742| /*
68743|     Note: If enabling debug_extension be careful not to
68744|     have many device add/removes, as everywhere where
68745|     | we
68746|     scan from PSMANDriverObject->DeviceObject there is
68747|     | the
68748|     possibility that it is still NULL (ie,
68749|     | IoCreateDevice)
68750|     was successful, but the MemAllocatePool hasnt
68751|     | happened yet.
68752|     This was seen on dual processor 800MHz with cluster
68753|     | enabled.
68754| */
68755| #ifdef DEBUG_EXTENSION
68756| __inline PVOID GetDeviceExtension(PDEVICE_OBJECT
68757|     | DevObj)
68758| {
68759|     PDEVICE_EXTENSION DE =
68760|     | (PDEVICE_EXTENSION)((PDEVICE_EXTENSION)(DevObj)->DeviceE
68761|     | xtension)->RealDeviceExtension;
68762| #if MEMDBG
68763|     MemCheckPool(DE);
68764| #endif
68765|     return DE;
68766| }
68767| #else
68768| #define GetDeviceExtension(DevObj)
68769|     | ((DevObj)->DeviceExtension)
68770| #endif
68771| #define GlobalData
68772|     | ((PSBPSMAN_EXTENSION)GetDeviceExtension(PSManObject))
68773|
68774|
68775| __inline PFILTERED_EXTENSION
68776|     | Inline_GetFilteredExtension ( PDEVICE_OBJECT DevObj
68777| #ifdef DEBUG
68778|     , const char *SourceFile, int SourceLine

```

```

68771| #endif /*DEBUG*/
68772| )
68773| {
68774|     ASSERT(DevObj != NULL);
68775|     PFILTERED_EXTENSION DevExt =
        | PFILTERED_EXTENSION(GetDeviceExtension(DevObj));
68776|     ASSERT(DevExt != NULL);
68777|
68778| #ifdef DEBUG
68779|     ULONG ObjectType =
        | PDEVICE_EXTENSION(DevExt)->ObjectType;
68780|     if ( ObjectType != OBJECT_FILTEREDDISK ) {
68781|         Debug(DEBUG_DCPSM,("!!! GetFilteredExtension:
        | Expected filtered extension (%d) but found object type
        | = %d\n",OBJECT_FILTEREDDISK,ObjectType));
68782|         Debug(DEBUG_DCPSM,("!!! DevObj=%08x, called
        | from %s(%d)\n",DevObj,SourceFile,SourceLine));
68783|         ASSERT(FALSE);
68784|     }
68785| #endif /*DEBUG*/
68786|
68787|     return DevExt;
68788| }
68789|
68790| __inline PVDISK_EXTENSION Inline_GetVDiskExtension (
        | PDEVICE_OBJECT DevObj
68791| #ifdef DEBUG
68792|     , const char *SourceFile, int SourceLine
68793| #endif /*DEBUG*/
68794| )
68795| {
68796|     ASSERT(DevObj != NULL);
68797|     PVDISK_EXTENSION DevExt = (PVDISK_EXTENSION)
        | GetDeviceExtension(DevObj);
68798|     ASSERT(DevExt != NULL);
68799|
68800| #ifdef DEBUG
68801|     ULONG ObjectType =
        | PDEVICE_EXTENSION(DevExt)->ObjectType;
68802|     if ( ObjectType != OBJECT_VIRTUALDISK ) {
68803|         Debug(DEBUG_DCPSM,("!!! GetVDiskExtension:
        | Expected vdisk extension (%d) but found object type =
        | %d\n",OBJECT_VIRTUALDISK,ObjectType));
68804|         Debug(DEBUG_DCPSM,("!!! DevObj=%08x, called
        | from %s(%d)\n",DevObj,SourceFile,SourceLine));
68805|         ASSERT(FALSE);
68806|     }
68807| #endif /*DEBUG*/
68808|
68809|     return DevExt;

```

```

68810| }
68811|
68812| #ifdef DEBUG
68813|     #define GetFilteredExtension(DevObj)
        | Inline_GetFilteredExtension((DevObj),__FILE__,__LINE__)
68814|     #define GetVDiskExtension(DevObj)
        | Inline_GetVDiskExtension((DevObj),__FILE__,__LINE__)
68815| #else /*DEBUG*/
68816|     #define GetFilteredExtension(DevObj)
        | Inline_GetFilteredExtension(DevObj)
68817|     #define GetVDiskExtension(DevObj)
        | Inline_GetVDiskExtension(DevObj)
68818| #endif /*DEBUG*/
68819|
68820|
68821| /*****
        | *****/
68822| *****/
        | *****/
68823| **
        | **
68824| ** To prevent deadlocks the following hierarchy must
        | be maintained!      **
68825| **
        | **
68826| ** SnapShot resource
        | **
68827| ** VDisk semaphore
        | **
68828| ** Global resource
        | **
68829| ** Rbtree resource
        | **
68830| **
        | **
68831| **
        | **
68832| **
        | **
68833| **
        | **
68834| **
        | **
68835| **
        | **
68836| **
        | **
68837| *****/
        | *****/
68838| *****/

```

```

| *****/
68839|
68840| /*
68841| resources need to be AT APC_LEVEL to be acquired
| contrary to what docs say
68842|
68843|
68844| From: "Brandon Allsop (Vlt Computer)"
| <a-branal@microsoft.com>
68845| To: 'Rob Green' <rgreen@cdp.com>
68846| Subject: RE: Driver Verifier
68847| Date: Thu, 5 Aug 1999 19:46:14 -0700
68848|
68849| Since APCs can occur at anytime it is a good idea to
| protect yourself when
68850| acquiring or releasing these resources. You can do this
| by either raising to
68851| irq! APC_LEVEL or by simply calling
| KeEnterCriticalRegion() before
68852| attempting the acquire or release.
68853|
68854|
68855| Checkout:
68856| KeEnterCriticalRegion()
68857| KeLeaveCriticalRegion()
68858| */
68859|
68860| #define MyAcquireResourceSharedLite(Resource,Wait) {
| \
68861| KeEnterCriticalRegion();
| \
68862| pmAcquireReaderLock(Resource,Wait); \
68863| KeLeaveCriticalRegion();
| \
68864| }
68865| #define MyReleaseResourceForThreadLite(Resource) { \
68866| KeEnterCriticalRegion();
| \
68867| /*lint -save -e746 -e740 */
| \
68868| pmReleaseReaderLock(Resource); \
68869| /*lint -restore */
| \
68870| KeLeaveCriticalRegion();
| \
68871| }
68872| #define MyAcquireResourceExclusiveLite(Resource,Wait) {
| \
68873| KeEnterCriticalRegion();
| \

```

```

68874|   pmAcquireWriterLock(Resource,Wait);   \
68875|   KeLeaveCriticalRegion();
68876|   | \
68877| }
68878| /*lint -emacro(740,ReleaseGlobalDeviceForRead) */
68879| /*lint -emacro(740,ReleaseGlobalDeviceForWrite) */
68880| /*lint -emacro(740,ReleaseDevExtForWrite) */
68881| /*lint -emacro(740,ReleaseDevExtForRead) */
68882| #define GetGlobalDeviceForRead()
68883| | MyAcquireResourceSharedLite (
68884| | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
68885| | DeviceResource, TRUE )
68886| #define ReleaseGlobalDeviceForRead()
68887| | MyReleaseResourceForThreadLite (
68888| | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
68889| | DeviceResource )
68890| #define GetGlobalDeviceForWrite()
68891| | MyAcquireResourceExclusiveLite(
68892| | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
68893| | DeviceResource, TRUE )
68894| #define ReleaseGlobalDeviceForWrite()
68895| | MyReleaseResourceForThreadLite(
68896| | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
68897| | DeviceResource )
68898| #define IsGlobalDeviceAcquiredForRead()
68899| | ExIsResourceAcquiredSharedLite(
68900| | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
68901| | DeviceResource)
68902| #define IsGlobalDeviceAcquiredForWrite()
68903| | pmRwLockedForWrite(&((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->DeviceResource)
68904| #define IsGlobalDeviceAcquired()
68905| | ((IsGlobalDeviceAcquiredForWrite()) ||
68906| | (IsGlobalDeviceAcquiredForRead()))
68907| #ifdef DEBUG_SNAPSHOTS
68908| // test to check for deadlocks
68909| #define IsSnapshotAcquiredForRead() 0
68910| #define IsSnapshotAcquiredForWrite()
68911| | (pmExamineSemaphore(&PSMSnapshotSemaphore)<1)
68912| #define IsSnapshotAcquired()
68913| | IsSnapshotAcquiredForWrite()
68914| #define GetNumSnapshotReaders() 0
68915| #define GetNumSnapshotWriters()
68916| | pmExamineSemaphore(&PSMSnapshotSemaphore) ? 0 : 1
68917| #define ReleaseSnapshotForRead()
68918| | ReleaseSnapshotResource()
68919| #define ReleaseSnapshotForWrite()

```

```

    | ReleaseSnapShotResource()
68900|
68901| #define GetSnapShotForRead()
    | AcquireSnapShotResource()
68902| #define GetSnapShotForWrite()
    | AcquireSnapShotResource()
68903|
68904|
68905| #else
68906| #define IsSnapShotAcquiredForRead()
    | ExIsResourceAcquiredSharedLite(
    | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
    | SnapShotResource )
68907| #define IsSnapShotAcquiredForWrite()
    | pmRwLockedForWrite(
    | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
    | SnapShotResource )
68908| #define IsSnapShotAcquired()
    | ((IsSnapShotAcquiredForWrite()) ||
    | (IsSnapShotAcquiredForRead()))
68909| #define GetNumSnapShotReaders()
    | pmRwLockNumReaders(
    | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
    | SnapShotResource )
68910| #define GetNumSnapShotWriters()
    | ExGetExclusiveWaiterCount(
    | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
    | SnapShotResource )
68911| #define ReleaseSnapShotForRead()
    | MyReleaseResourceForThreadLite (
    | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
    | SnapShotResource )
68912| #define ReleaseSnapShotForWrite()
    | MyReleaseResourceForThreadLite(
    | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
    | SnapShotResource )
68913|
68914| #ifdef DEBUG
68915|     #define GetSnapShotForRead()
    | if(IsSnapShotAcquired()) {
    | \
68916|     | Debug(DEBUG_DCPSM,("GetSnapShotForRead: SnapShot is
    | already acquired!\n")); \
68917|     }
    | \
68918|
    | MyAcquireResourceSharedLite (
    | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
    | SnapShotResource, TRUE )

```



```

68919|
68920|  #define GetSnapShotForWrite()
        | if(!sSnapShotAcquired()) {
        | \
68921|
        | Debug(DEBUG_DCPSM,("GetSnapShotForWrite: SnapShot is
        | already acquired!\n")); \
68922|
        | DbgBreakPoint();
        | \
68923|                                     }
        | \
68924|
        | MyAcquireResourceExclusiveLite(
        | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
        | SnapShotResource, TRUE )
68925| #else
68926|  #define GetSnapShotForRead()
        | MyAcquireResourceSharedLite (
        | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
        | SnapShotResource, TRUE )
68927|  #define GetSnapShotForWrite()
        | MyAcquireResourceExclusiveLite(
        | &((PSBPSMAN_EXTENSION)GetDeviceExtension(PsManObject))->
        | SnapShotResource, TRUE )
68928| #endif // debug
68929| #endif // DEBUG_SNAPSHOTS
68930|
68931| #define GetDevExtForRead(DevExt)
        | MyAcquireResourceSharedLite (
        | &(DevExt)->DeviceExtResource, TRUE )
68932| #define ReleaseDevExtForRead(DevExt)
        | MyReleaseResourceForThreadLite (
        | &(DevExt)->DeviceExtResource )
68933| #define GetDevExtForWrite(DevExt)
        | MyAcquireResourceExclusiveLite(
        | &(DevExt)->DeviceExtResource, TRUE )
68934| #define ReleaseDevExtForWrite(DevExt)
        | MyReleaseResourceForThreadLite(
        | &(DevExt)->DeviceExtResource )
68935|
68936|
68937| #define PsmGetObjectTypes(DevObj)
        | (((PDEVICE_EXTENSION)(GetDeviceExtension(DevObj)))->Obje
        | ctTypes)
68938| #define GetDriveNumFromObject(DevObj)
        | ((GetFilteredExtension(DevObj)->DiskNumber)
68939|
68940| #ifdef DEBUG
68941| #define FREE_POINTER(_ptr_) { ASSERT((_ptr_)!=NULL);

```

```

    | MemFreePool((_ptr_)); (_ptr_)=NULL; }
68942| #else
68943| #define FREE_POINTER(_ptr_) { MemFreePool((_ptr_));
    | (_ptr_)=NULL; }
68944| #endif
68945|
68946| //
68947| // The following macros are used to establish the
    | semantics needed
68948| // to do a return from within a try-finally clause.
    | As a rule every
68949| // try clause must end with a label call try_exit.
    | For example,
68950| //
68951| //     try {
68952| //         :
68953| //         :
68954| //
68955| //     try_exit: NOTHING;
68956| //     } finally {
68957| //
68958| //         :
68959| //         :
68960| //     }
68961| //
68962| // Every return statement executed inside of a try
    | clause should use the
68963| // try_return macro. If the compiler fully supports
    | the try-finally construct
68964| // then the macro should be
68965| //
68966| //     #define try_return(S) { return(S); }
68967| //
68968| // If the compiler does not support the try-finally
    | construct then the macro
68969| // should be
68970| //
68971| //     #define try_return(S) { S; goto try_exit; }
68972| //
68973|
68974| /*lint -emacro(665,try_return) */
68975| /*lint -emacro(527,try_return) */
68976|
68977| #define try_return(S) { S; goto try_exit; }
68978|
68979| #define
    | MyInterlockedRemoveEntryList(SpinLock,ListEntry) { \
68980|     KIRQL _oldIrql_;
    | \
68981|     pmAcquireSpinLock ( SpinLock, &_oldIrql_ );

```

```

| \
68982| RemoveEntryList(ListEntry);
| \
68983| pmReleaseSpinLock( SpinLock, _oldIrql_ );
| \
68984| }
68985|
68986|
68987| #define NOT_REFERENCED(x) /*lint -esym(715,x) */
68988|
68989| #ifdef DEBUG
68990| #define MAX_BACK_LOG_FOR_DEBUG 26
68991| #endif
68992|
68993|
68994| void DebugLogThread( PVOID Context );
68995|
68996| #if _WIN32_WINNT >= 0x0500
68997|
68998| NTSTATUS
68999| PSMANAddDevice(
69000|     IN PDRIVER_OBJECT DriverObject,
69001|     IN PDEVICE_OBJECT PhysicalDeviceObject
69002| );
69003|
69004| WMIGUIDREGINFO PSMANGuidList[];
69005|
69006| #define PSMANGuidCount (sizeof(PSMANGuidList) /
    | sizeof(WMIGUIDREGINFO))
69007|
69008| #else
69009| #define IoCopyCurrentIrpStackLocationToNext( Irp ) { \
69010|     PIO_STACK_LOCATION irpSp; \
69011|     PIO_STACK_LOCATION nextIrpSp; \
69012|     irpSp = IoGetCurrentIrpStackLocation( (Irp) ); \
69013|     nextIrpSp = IoGetNextIrpStackLocation( (Irp) ); \
69014|     RtlCopyMemory( nextIrpSp, irpSp,
    | FIELD_OFFSET(IO_STACK_LOCATION, CompletionRoutine)); \
69015|     nextIrpSp->Control = 0; }
69016|
69017|
69018| #endif
69019|
69020| #define DN_MakePointer(Start,Offset)
    | ((PVOID)((((char*)(Start))+(Offset)))
69021| #define DN_MakeOffset(Start,Offset)
    | ((ULONG)((((char*)(Offset))-((char*)(Start))))
69022|
69023|
69024| #define PSM_BOOT_STAGE 0

```

```

69025| #define PSM_INIT_STAGE 1
69026| #define PSM_NORMAL_STAGE 10
69027|
69028| #define max(a,b) ((a)>(b) ? (a) : (b))
69029| #define min(a,b) ((a)<(b) ? (a) : (b))
69030|
69031| #define INVALID_HANDLE_VALUE ((HANDLE)(-1))
69032|
69033| typedef USHORT      WORD;
69034|
69035| extern ULONG gVDiskDoVirtualIO;
69036|
69037| extern GLOBALTYPE ULONG GlobalVersionMajor;
69038| extern GLOBALTYPE ULONG GlobalVersionMinor;
69039| extern GLOBALTYPE ULONG GlobalBuildNumber;
69040|
69041| extern GLOBALTYPE PEPROCESS GlobalSystemProcessId;
69042|
69043| #define PSM_PRIVATE_DIR L"Persistent Storage Manager
    | State"
69044| #define PSM_PRIVATE_DIR_SLASH PSM_PRIVATE_DIR L"\\"
69045|
69046| typedef struct sPSM_GetPSMEventEntry {
69047|     pPSM_GetPSMEvent Event;
69048|     LIST_ENTRY ListEntry;
69049|     PIRP Irp;
69050| } tPSM_GetPSMEventEntry,*pPSM_GetPSMEventEntry;
69051|
69052|
69053|
69054| File Listing: SECURITY.cpp
69055|
69056| #include "precomp.h"
69057|
69058| /*
69059|     Portions Copyright 2000 Mark Russinovich
        | www.sysinternals.com
69060|
69061| > Mark,
69062| > I want to use portions of secsys (from the
        | Information page, Device
69063| > Object Security, Secdsrc.zip or link
69064| > http://www.sysinternals.com/devsec.htm)
69065| > in a driver i have written.
69066| >
69067| > More specifically i want to use the secsys.h header
        | file (list the
69068| > definitions for the NT security APIs).
69069| > I also want to use some functions from secsys.c. In
        | particular

```

```

69070| > NTIRemoveWorldAce and its support functions.
69071| >
69072| > I couldn't find information on licensing on your web
    | page other than what
69073| > appears to be directed to your utilites. I would
    | like to know if
69074| > i can use
69075| > the above mentioned code for inclusion in a
    | commercial product. I would
69076| > retain your copyrights in the source code of course..
69077| >
69078|
69079| Yes, you have permission to include code from secsys in
    | your project free of
69080| any licensing fees.
69081|
69082| -Mark
69083|
69084| Coauthor, "Inside Windows 2000, 3rd Ed."
69085| Contributing Editor, "Windows 2000 Magazine"
69086| Chief Software Architect, Winternals Software
69087|
69088| */
69089|
69090| //-----
    | -----
69091| //
69092| // NTIDeleteAce
69093| //
69094| // The NT Kernel does not implement a DeleteAce
    | function so we
69095| // have to create our own.
69096| //
69097| //-----
    | -----
69098| BOOLEAN NTIDeleteAce( PACL Acl, USHORT AceIndex )
69099| {
69100|     PACE_HEADER aceHeader;
69101|     WORD aceSize;
69102|     USHORT i;
69103|
69104|     //
69105|     // Sanity check: does the ACL contain at least
    | AceIndex ACEs?
69106|     //
69107|     if( AceIndex > Acl->AceCount ) return FALSE;
69108|
69109|     aceHeader = (PACE_HEADER) ((PUCHAR) Acl +
    | sizeof(ACL));
69110|     for( i = 0; i < AceIndex; i++ ) {

```

```

69111|
69112|     aceHeader = (PACE_HEADER) ((PUCHAR) aceHeader +
    | aceHeader->AceSize);
69113| }
69114|
69115| //
69116| // Now subtract the ace size from the header, and
    | shift
69117| // all aces to the right of this one over to fill the
    | hole
69118| //
69119| aceSize = aceHeader->AceSize;
69120|
69121| RtlMoveMemory( aceHeader, ((PUCHAR) aceHeader +
    | aceSize),
69122|     Acl->AclSize - (((PUCHAR) aceHeader - (PUCHAR)
    | Acl) + aceSize));
69123| Acl->AclSize -= aceSize;
69124| Acl->AceCount--;
69125| return TRUE;
69126| }
69127|
69128|
69129| //-----
    | -----
69130| //
69131| // NTIGetAce
69132| //
69133| // The NT Kernel does not implement a GetAce function
    | so we
69134| // have to create our own.
69135| //
69136| //-----
    | -----
69137| PACE_HEADER NTIGetAce( PACL Acl, USHORT AceIndex )
69138| {
69139|     PACE_HEADER    aceHeader;
69140|     USHORT         i;
69141|
69142|     //
69143|     // Sanity check: does the ACL contain at least
    | AceIndex ACEs?
69144|     //
69145|     if( AceIndex > Acl->AceCount ) return NULL;
69146|
69147|     aceHeader = (PACE_HEADER) ((PUCHAR) Acl +
    | sizeof(ACL));
69148|     for( i = 0; i < AceIndex; i++ ) {
69149|
69150|         aceHeader = (PACE_HEADER) ((PUCHAR) aceHeader +

```

```

    | aceHeader->AceSize);
69151| }
69152|
69153| return aceHeader;
69154| }
69155|
69156|
69157| //-----
    | -----
69158| //
69159| // NTIMakeAbsoluteSD
69160| //
69161| // Takes a self-relative security descriptor and
    | returns an allocated
69162| // absolute version. Caller is responsible for freeing
    | the allocated
69163| // buffer on success.
69164| //
69165| //-----
    | -----
69166| NTSTATUS NTIMakeAbsoluteSD( PSECURITY_DESCRIPTOR
    | RelSecurityDescriptor,
69167|         PSECURITY_DESCRIPTOR
    | *pAbsSecurityDescriptor )
69168| {
69169|     NTSTATUS    status;
69170|     BOOLEAN     DaclPresent, DaclDefaulted,
    | OwnerDefaulted, GroupDefaulted;
69171|     PACL        Dacl;
69172|     PSID        Owner, Group;
69173|     PSECURITY_DESCRIPTOR absSecurityDescriptor;
69174|
69175|     //
69176|     // Initialize buffer pointers
69177|     //
69178|     absSecurityDescriptor = (PSECURITY_DESCRIPTOR)
    | ExAllocatePool( NonPagedPool, 1024 );
69179|     *pAbsSecurityDescriptor = absSecurityDescriptor;
69180|
69181|     //
69182|     // Create an absolute-form security descriptor for
    | manipulation.
69183|     // The one on the security descriptor is in
    | self-relative form.
69184|     //
69185|     status = RtlCreateSecurityDescriptor(
    | absSecurityDescriptor,
69186|         SECURITY_DESCRIPTOR_REVISION );
69187|     if( !NT_SUCCESS( status ) ) {
69188|

```

```

69189|    DbgPrint(("Secsys: Unable to initialize security
        | descriptor\n"));
69190|    goto cleanup;
69191| }
69192|
69193| //
69194| // Locate the descriptor's DACL and apply the DACL
        | to the new
69195| // descriptor we're going to modify
69196| //
69197| status = RtlGetDaclSecurityDescriptor(
        | RelSecurityDescriptor,
69198|          &DaclPresent,
69199|          &Dacl,
69200|          &DaclDefaulted );
69201| if( !NT_SUCCESS( status ) || !DaclPresent ) {
69202|
69203|     if( !NT_SUCCESS( status ) ) {
69204|
69205|         Debug(DEBUG_INIT,("Secsys: Error obtaining
            | security descriptor's DACL: %x\n", status ));
69206|
69207|     } else {
69208|
69209|         Debug(DEBUG_INIT,("Secsys: Security descriptor
            | does not have a DACL\n" ));
69210|     }
69211|
69212|     goto cleanup;
69213| }
69214|
69215| status = RtlSetDaclSecurityDescriptor(
        | absSecurityDescriptor,
69216|          DaclPresent,
69217|          Dacl,
69218|          DaclDefaulted );
69219| if( !NT_SUCCESS( status ) ) {
69220|
69221|     Debug(DEBUG_INIT,("Secsys: Coult not set new
            | security descriptor DACL: %x\n", status ));
69222|     goto cleanup;
69223| }
69224|
69225| //
69226| // We would get and apply the SACL at this point,
        | but NT does not export
69227| // the appropriate function,
        | RtlGetSaclSecurityDescriptor :(
69228| //
69229|

```



```

69230| //
69231| // Get and apply the owner
69232| //
69233| status = RtlGetOwnerSecurityDescriptor(
        | RelSecurityDescriptor,
69234|          &Owner,
69235|          &OwnerDefaulted );
69236| if( !NT_SUCCESS( status ) ) {
69237|
69238|     Debug(DEBUG_INIT,("Secsys: Could not security
        | descriptor owner: %x\n", Owner ));
69239|     goto cleanup;
69240| }
69241|
69242| status = RtlSetOwnerSecurityDescriptor(
        | absSecurityDescriptor,
69243|          Owner,
69244|          OwnerDefaulted );
69245|
69246| if( !NT_SUCCESS( status ) ) {
69247|
69248|     Debug(DEBUG_INIT,("Secsys: Could not set owner:
        | %x\n", status ));
69249|     goto cleanup;
69250| }
69251|
69252| //
69253| // Get and apply group
69254| //
69255| status = RtlGetGroupSecurityDescriptor(
        | RelSecurityDescriptor,
69256|          &Group,
69257|          &GroupDefaulted );
69258| if( !NT_SUCCESS( status ) ) {
69259|
69260|     Debug(DEBUG_INIT,("Secsys: Could not security
        | descriptor group: %x\n", Owner ));
69261|     goto cleanup;
69262| }
69263|
69264| status = RtlSetGroupSecurityDescriptor(
        | absSecurityDescriptor,
69265|          Group,
69266|          GroupDefaulted );
69267|
69268| if( !NT_SUCCESS( status ) ) {
69269|
69270|     Debug(DEBUG_INIT,("Secsys: Could not set group:
        | %x\n", status ));
69271|     goto cleanup;

```

```

69272| }
69273|
69274| //
69275| // Finally, make sure that what we made is valid
69276| //
69277| if( !RtlValidSecurityDescriptor(
    | absSecurityDescriptor )) {
69278|
69279|     Debug(DEBUG_INIT,("Secsys: absolute descriptor not
    | valid!\n"));
69280|     status = STATUS_UNSUCCESSFUL;
69281| }
69282|
69283| //
69284| // Done! Return.
69285| //
69286|
69287| cleanup:
69288|
69289| if( !NT_SUCCESS( status ) ) {
69290|
69291|     ExFreePool( absSecurityDescriptor );
69292| }
69293| return status;
69294| }
69295|
69296|
69297| //-----
    | -----
69298| //
69299| // NTIRemoveWorldAce
69300| //
69301| // Scans the passed security descriptor's DACL, looking
    | for
69302| // the World SID's ACE (its first because the of the
    | way device object
69303| // security descriptors are created) and removes it.
69304| //
69305| // If successful, the original security descriptor is
    | deallocated
69306| // and a new one is returned.
69307| //
69308| //-----
    | -----
69309| NTSTATUS NTIRemoveWorldAce( PSECURITY_DESCRIPTOR
    | SecurityDescriptor,
69310|     PSECURITY_DESCRIPTOR
    | *NewSecurityDescriptor )
69311| {
69312|     PSECURITY_DESCRIPTOR     absSecurityDescriptor;

```

```

69313| PSECURITY_DESCRIPTOR    relSecurityDescriptor;
69314| PACE_HEADER              aceHeader;
69315| NTSTATUS                status;
69316| PACL                    Dacl;
69317| BOOLEAN                DaclPresent, DaclDefaulted;
69318| USHORT                  aceIndex;
69319| ULONG                   worldSidLength;
69320| SID_IDENTIFIER_AUTHORITY worldSidAuthority =
    | SECURITY_WORLD_SID_AUTHORITY;
69321| PULONG                  worldSidSubAuthority;
69322| ULONG                   relLength;
69323| PSID                    worldSid;
69324| PSID                    aceSid;
69325|
69326| //
69327| // First, get an absolute version of the
    | self-relative descriptor
69328| //
69329| relLength = RtlLengthSecurityDescriptor(
    | SecurityDescriptor );
69330| status = NTMakeAbsoluteSD( SecurityDescriptor,
69331|                            &absSecurityDescriptor );
69332| if( !NT_SUCCESS( status ) ) {
69333|
69334|     return status;
69335| }
69336|
69337| //
69338| // Pull the DACL out so that we can scan it
69339| //
69340| status = RtlGetDaclSecurityDescriptor(
    | absSecurityDescriptor,
69341|    &DaclPresent,
69342|    &Dacl,
69343|    &DaclDefaulted );
69344| if( !NT_SUCCESS( status ) || !DaclPresent ) {
69345|
69346|     Debug(DEBUG_INIT,("Secsys: strange - couldn't get
    | DACL from our absolute SD: %x\n",
69347|        status ));
69348|     ExFreePool( absSecurityDescriptor );
69349|     return status;
69350| }
69351|
69352| //
69353| // Initialize a SID that identifies the
    | world-authority so
69354| // that we can recognize it in the ACL
69355| //
69356| worldSidLength = RtlLengthRequiredSid( 1 );

```

```

69357| worldSid = (PSID) ExAllocatePool( PagedPool,
    | worldSidLength );
69358| RtlInitializeSid( worldSid, &worldSidAuthority, 1 );
69359| worldSidSubAuthority = RtlSubAuthoritySid( worldSid,
    | 0 );
69360| *worldSidSubAuthority = SECURITY_WORLD_RID;
69361|
69362| //
69363| // Now march through the ACEs looking for the World
    | ace. We could
69364| // do one of two things:
69365| //
69366| // - remove the ACE
69367| // - convert it into a grant-nothing ACE
69368| //
69369| // For demonstration purposes I'll remove the ACE.
    | In addition,
69370| // this requires that I implement kernel-mode GetAce
    | and DeleteAce functions,
69371| // since they are not implemented by the NT kernel.
69372| //
69373| Debug(DEBUG_INIT,("Secsys: %d ACEs in DACL\n",
    | Dacl->AceCount ));
69374|
69375| for( aceIndex = 0; aceIndex < Dacl->AceCount;
    | aceIndex++ ) {
69376|
69377|     aceHeader = NTIGetAce( Dacl, aceIndex );
69378|
69379|     Debug(DEBUG_INIT,(" ACE: type: %s mask: %x\n",
69380|         (aceHeader->AceType & ACCESS_DENIED_ACE_TYPE
    | ? "Deny" : "Allow"),
69381|         *(PULONG) ((PUCHAR) aceHeader +
    | sizeof(ACE_HEADER))));
69382|
69383|     //
69384|     // Get the SID in this ACE and see if its the
    | WORLD (Everyone) SID
69385|     //
69386|     aceSid = (PSID) ((PUCHAR) aceHeader +
    | sizeof(ACE_HEADER) + sizeof(ULONG));
69387|     if( RtlEqualSid( worldSid, aceSid )) {
69388|
69389|         //
69390|         // We found it: remove it.
69391|         //
69392|         Debug(DEBUG_INIT,("Secsys: Deleting ace %d\n",
    | aceIndex ));
69393|         NTIDeleteAce( Dacl, aceIndex );
69394|         break;

```

```

69395|    }
69396|    }
69397|
69398|    //
69399|    // Write new DACL back to security descriptor
69400|    //
69401|    status = RtlSetDaclSecurityDescriptor(
        | absSecurityDescriptor,
69402|        TRUE,
69403|        Dacl,
69404|        FALSE );
69405|    if( !NT_SUCCESS( status )) {
69406|
69407|        Debug(DEBUG_INIT,("Secsys: Could not update SD
        | Dacl: %x\n", status ));
69408|        goto cleanup;
69409|    }
69410|
69411|    //
69412|    // Make sure its valid
69413|    //
69414|    if( !RtlValidSecurityDescriptor(
        | absSecurityDescriptor )) {
69415|
69416|        Debug(DEBUG_INIT,("Secsys: SD after remove is
        | invalid!\n"));
69417|        status = STATUS_UNSUCCESSFUL;
69418|        goto cleanup;
69419|    }
69420|
69421|    //
69422|    // Now convert the security descriptor back to
69423|    // self-relative
69424|    //
69425|    relSecurityDescriptor = ExAllocatePool( PagedPool,
        | relLength );
69426|    status = RtlAbsoluteToSelfRelativeSD(
        | absSecurityDescriptor,
69427|        relSecurityDescriptor, &relLength
        | );
69428|    if( !NT_SUCCESS( status )) {
69429|
69430|        Debug(DEBUG_INIT,("Could not convert absolute SD
        | to relative: %x\n", status ));
69431|    }
69432|
69433|    //
69434|    // Final step, free the original security descriptor
        | and return the new one
69435|    //

```

```

69436|  ExFreePool( SecurityDescriptor );
69437|  *NewSecurityDescriptor = relSecurityDescriptor;
69438|
69439| cleanup:
69440|  ExFreePool( worldSid );
69441|  ExFreePool( absSecurityDescriptor );
69442|  return status;
69443| }
69444|
69445|
69446|
69447| File Listing: SECURITY.h
69448|
69449| //=====
    | =====
69450| //
69451| // Secsys.h
69452| //
69453| // Copyright (C) 1998 Mark Russinovich
69454| //
69455| // Security-related definitions and APIs not included
    | in NTDDK.H.
69456| // Note that only the definitions required for Secsys
    | are included
69457| // here.
69458| //
69459| //=====
    | =====
69460|
69461| /*
69462|  Portions Copyright 2000 Mark Russinovich
    | www.sysinternals.com
69463|
69464| > Mark,
69465| > I want to use portions of secsys (from the
    | Information page, Device
69466| > Object Security, Secdsrc.zip or link
69467| > http://www.sysinternals.com/devsec.htm)
69468| > in a driver i have written.
69469| >
69470| > More specifically i want to use the secsys.h header
    | file (list the
69471| > definitions for the NT security APIs).
69472| > I also want to use some functions from secsys.c. In
    | particular
69473| > NTIRemoveWorldAce and its support functions.
69474| >
69475| > I couldn't find information on licensing on your web
    | page other than what
69476| > appears to be directed to your utilites. I would

```

```

| like to know if
69477| > i can use
69478| > the above mentioned code for inclusion in a
| commercial product. I would
69479| > retain your copyrights in the source code of course..
69480| >
69481|
69482| Yes, you have permission to include code from secsys in
| your project free of
69483| any licensing fees.
69484|
69485| -Mark
69486|
69487| Coauthor, "Inside Windows 2000, 3rd Ed."
69488| Contributing Editor, "Windows 2000 Magazine"
69489| Chief Software Architect, Winternals Software
69490|
69491| */
69492|
69493|
69494| //
69495| // Types for Win32 header definition conversion
69496| //
69497| //typedef UCHAR      BYTE;
69498| //typedef ULONG      DWORD;
69499| //typedef USHORT     WORD;
69500|
69501| //
69502| // Security definitions needed for SecSys
69503| //
69504|
69505| typedef struct _ACE_HEADER {
69506|     BYTE AceType;
69507|     BYTE AceFlags;
69508|     WORD AceSize;
69509| } ACE_HEADER;
69510| typedef ACE_HEADER *PACE_HEADER;
69511|
69512| //
69513| // The following are the predefined ace types that go
| into the AceType
69514| // field of an Ace header.
69515| //
69516|
69517| #define ACCESS_ALLOWED_ACE_TYPE      (0x0)
69518| #define ACCESS_DENIED_ACE_TYPE      (0x1)
69519| #define SYSTEM_AUDIT_ACE_TYPE      (0x2)
69520| #define SYSTEM_ALARM_ACE_TYPE      (0x3)
69521|
69522|

```

```

69523| typedef struct _SID_IDENTIFIER_AUTHORITY {
69524|     BYTE Value[6];
69525| } SID_IDENTIFIER_AUTHORITY, *PSID_IDENTIFIER_AUTHORITY;
69526|
69527| #define SECURITY_WORLD_SID_AUTHORITY    {0,0,0,0,0,1}
69528| #define SECURITY_WORLD_RID              (0x00000000L)
69529|
69530| #if 0 // in NTDDK.H
69531|
69532| // Win32: InitializeSecurityDescriptor
69533| NTSTATUS
69534| NTAPI
69535| RtlCreateSecurityDescriptor(
69536|     PSECURITY_DESCRIPTOR SecurityDescriptor,
69537|     ULONG Revision
69538| );
69539|
69540| // Win32: IsValidSecurityDescriptor
69541| BOOLEAN
69542| NTAPI
69543| RtlValidSecurityDescriptor(
69544|     PSECURITY_DESCRIPTOR SecurityDescriptor
69545| );
69546|
69547| // Win32: GetSecurityDescriptorLength
69548| ULONG
69549| NTAPI
69550| RtlLengthSecurityDescriptor(
69551|     PSECURITY_DESCRIPTOR SecurityDescriptor
69552| );
69553|
69554| // Win32: SetSecurityDescriptorDacl
69555| NTSTATUS
69556| NTAPI
69557| RtlSetDaclSecurityDescriptor(
69558|     PSECURITY_DESCRIPTOR SecurityDescriptor,
69559|     BOOLEAN DaclPresent,
69560|     PACL Dacl,
69561|     BOOLEAN DaclDefaulted
69562| );
69563|
69564| #endif // in NTDDK.H
69565|
69566| // Win32: GetSecurityDescriptorDacl
69567| NTSTATUS
69568| NTAPI
69569| RtlGetDaclSecurityDescriptor(
69570|     PSECURITY_DESCRIPTOR SecurityDescriptor,
69571|     PBOOLEAN DaclPresent,
69572|     PACL *Dacl,

```



```

69573| PBOOLEAN DaclDefaulted
69574| );
69575|
69576| // Win32: SetSecurityDescriptorSacl
69577| //
69578| // NOTE: NT does not export
        | RtlGetSaclSecurityDescriptor!
69579| NTSTATUS
69580| NTAPI
69581| RtlSetSaclSecurityDescriptor(
69582|     PSECURITY_DESCRIPTOR SecurityDescriptor,
69583|     BOOLEAN SaclPresent,
69584|     PACL Sacl,
69585|     BOOLEAN SaclDefaulted
69586| );
69587|
69588| // Win32: SetSecurityDescriptorOwner
69589| NTSTATUS
69590| NTAPI
69591| RtlSetOwnerSecurityDescriptor(
69592|     PSECURITY_DESCRIPTOR SecurityDescriptor,
69593|     PSID Owner,
69594|     BOOLEAN OwnerDefaulted
69595| );
69596|
69597| // Win32: GetSecurityDescriptorOwner
69598| NTSTATUS
69599| NTAPI
69600| RtlGetOwnerSecurityDescriptor(
69601|     PSECURITY_DESCRIPTOR SecurityDescriptor,
69602|     PSID *Owner,
69603|     PBOOLEAN OwnerDefaulted
69604| );
69605|
69606| // Win32: SetSecurityDescriptorGroup
69607| NTSTATUS
69608| NTAPI
69609| RtlSetGroupSecurityDescriptor(
69610|     PSECURITY_DESCRIPTOR SecurityDescriptor,
69611|     PSID Group,
69612|     BOOLEAN GroupDefaulted
69613| );
69614|
69615| // Win32: GetSecurityDescriptorGroup
69616| NTSTATUS
69617| NTAPI
69618| RtlGetGroupSecurityDescriptor(
69619|     PSECURITY_DESCRIPTOR SecurityDescriptor,
69620|     PSID *Group,
69621|     PBOOLEAN GroupDefaulted

```

```
69622| );
69623|
69624|
69625| // Win32: GetSidLengthRequired
69626| ULONG
69627| NTAPI
69628| RtlLengthRequiredSid(
69629|     UCHAR SubAuthorityCount
69630| );
69631|
69632|
69633| // Win32: InitializeSid
69634| NTSTATUS
69635| NTAPI
69636| RtlInitializeSid(
69637|     PSID Sid,
69638|     PSID_IDENTIFIER_AUTHORITY pIdentifierAuthority,
69639|     UCHAR nSubAuthorityCount
69640| );
69641|
69642|
69643| // Win32: GetSidSubAuthority
69644| PULONG
69645| NTAPI
69646| RtlSubAuthoritySid(
69647|     PSID pSid,
69648|     ULONG nSubAuthority
69649| );
69650|
69651|
69652| // Win32: IsEqualSid
69653| BOOLEAN
69654| NTAPI
69655| RtlEqualSid(
69656|     PSID Sid1,
69657|     PSID Sid2
69658| );
69659|
69660|
69661| // Win32: MakeSelfRelativeSD
69662| NTSTATUS
69663| NTAPI
69664| RtlAbsoluteToSelfRelativeSD(
69665|     PSECURITY_DESCRIPTOR AbsoluteSecurityDescriptor,
69666|     PSECURITY_DESCRIPTOR
        | SelfRelativeSecurityDescriptor,
69667|     PULONG BufferLength
69668| );
69669|
69670|
```

```

69671|
69672|
69673|
69674| NTSTATUS NTIRemoveWorldAce( PSECURITY_DESCRIPTOR
    | SecurityDescriptor,
69675|         PSECURITY_DESCRIPTOR
    | *NewSecurityDescriptor );
69676|
69677|
69678|
69679| File Listing: sfilter.cpp
69680|
69681| #define DO_FILE_SYSTEM_FILTER 1
69682| #include "precomp.h"
69683|
69684| #if DO_FILE_SYSTEM_FILTER
69685|
69686|     #define FSCTL_REQUEST_OPLOCK_LEVEL_1
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 0, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69687|     #define FSCTL_REQUEST_OPLOCK_LEVEL_2
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 1, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69688|     #define FSCTL_REQUEST_BATCH_OPLOCK
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 2, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69689|     #define FSCTL_OPLOCK_BREAK_ACKNOWLEDGE
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 3, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69690|     #define FSCTL_OPBATCH_ACK_CLOSE_PENDING
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 4, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69691|     #define FSCTL_OPLOCK_BREAK_NOTIFY
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 5, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69692|     #define FSCTL_LOCK_VOLUME
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 6, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69693|     #define FSCTL_UNLOCK_VOLUME
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 7, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69694|     #define FSCTL_DISMOUNT_VOLUME
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 8, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69695| // decommissioned fsctl value
        | 9
69696|     #define FSCTL_IS_VOLUME_MOUNTED
        | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 10, METHOD_BUFFERED,
        | FILE_ANY_ACCESS)
69697|     #define FSCTL_IS_PATHNAME_VALID

```

```

| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 11, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // PATHNAME_BUFFER,
69698| #define FSCTL_MARK_VOLUME_DIRTY
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 12, METHOD_BUFFERED,
| FILE_ANY_ACCESS)
69699| // decommissioned fsctl value
| 13
69700| #define FSCTL_QUERY_RETRIEVAL_POINTERS
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 14, METHOD_NEITHER,
| FILE_ANY_ACCESS)
69701| #define FSCTL_GET_COMPRESSION
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 15, METHOD_BUFFERED,
| FILE_ANY_ACCESS)
69702| #define FSCTL_SET_COMPRESSION
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 16, METHOD_BUFFERED,
| FILE_READ_DATA | FILE_WRITE_DATA)
69703| // decommissioned fsctl value
| 17
69704| // decommissioned fsctl value
| 18
69705| #define FSCTL_MARK_AS_SYSTEM_HIVE
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 19, METHOD_NEITHER,
| FILE_ANY_ACCESS)
69706| #define FSCTL_OPLOCK_BREAK_ACK_NO_2
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 20, METHOD_BUFFERED,
| FILE_ANY_ACCESS)
69707| #define FSCTL_INVALIDATE_VOLUMES
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 21, METHOD_BUFFERED,
| FILE_ANY_ACCESS)
69708| #define FSCTL_QUERY_FAT_BPB
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 22, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // FSCTL_QUERY_FAT_BPB_BUFFER
69709| #define FSCTL_REQUEST_FILTER_OPLOCK
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 23, METHOD_BUFFERED,
| FILE_ANY_ACCESS)
69710| #define FSCTL_FILESYSTEM_GET_STATISTICS
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 24, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // FILESYSTEM_STATISTICS
69711| #if(_WIN32_WINNT >= 0x0400)
69712| #define FSCTL_GET_NTFS_VOLUME_DATA
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 25, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // NTFS_VOLUME_DATA_BUFFER
69713| #define FSCTL_GET_NTFS_FILE_RECORD
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 26, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // NTFS_FILE_RECORD_INPUT_BUFFER,
| NTFS_FILE_RECORD_OUTPUT_BUFFER
69714| #define FSCTL_GET_VOLUME_BITMAP
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 27, METHOD_NEITHER,
| FILE_ANY_ACCESS) // STARTING_LCN_INPUT_BUFFER,
| VOLUME_BITMAP_BUFFER

```

```

69715|     #define FSCTL_GET_RETRIEVAL_POINTERS
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 28, METHOD_NEITHER,
| FILE_ANY_ACCESS) // STARTING_VCN_INPUT_BUFFER,
| RETRIEVAL_POINTERS_BUFFER
69716|     #define FSCTL_MOVE_FILE
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 29, METHOD_BUFFERED,
| FILE_SPECIAL_ACCESS) // MOVE_FILE_DATA,
69717|     #define FSCTL_IS_VOLUME_DIRTY
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 30, METHOD_BUFFERED,
| FILE_ANY_ACCESS)
69718|     #define FSCTL_GET_HFS_INFORMATION
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 31, METHOD_BUFFERED,
| FILE_ANY_ACCESS)
69719|     #define FSCTL_ALLOW_EXTENDED_DASD_IO
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 32, METHOD_NEITHER,
| FILE_ANY_ACCESS)
69720|     #endif /* _WIN32_WINNT >= 0x0400 */
69721|
69722|     #if(_WIN32_WINNT >= 0x0500)
69723|     #define FSCTL_READ_PROPERTY_DATA
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 33, METHOD_NEITHER,
| FILE_ANY_ACCESS)
69724|     #define FSCTL_WRITE_PROPERTY_DATA
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 34, METHOD_NEITHER,
| FILE_ANY_ACCESS)
69725|     #define FSCTL_FIND_FILES_BY_SID
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 35, METHOD_NEITHER,
| FILE_ANY_ACCESS)
69726| // decommissioned fsctl value
| 36
69727|     #define FSCTL_DUMP_PROPERTY_DATA
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 37, METHOD_NEITHER,
| FILE_ANY_ACCESS)
69728|     #define FSCTL_SET_OBJECT_ID
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 38, METHOD_BUFFERED,
| FILE_SPECIAL_ACCESS) // FILE_OBJECTID_BUFFER
69729|     #define FSCTL_GET_OBJECT_ID
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 39, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // FILE_OBJECTID_BUFFER
69730|     #define FSCTL_DELETE_OBJECT_ID
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 40, METHOD_BUFFERED,
| FILE_SPECIAL_ACCESS)
69731|     #define FSCTL_SET_REPARSE_POINT
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 41, METHOD_BUFFERED,
| FILE_SPECIAL_ACCESS) // REPARSE_DATA_BUFFER,
69732|     #define FSCTL_GET_REPARSE_POINT
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 42, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // REPARSE_DATA_BUFFER
69733|     #define FSCTL_DELETE_REPARSE_POINT
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 43, METHOD_BUFFERED,

```

```

| FILE_SPECIAL_ACCESS) // REPARSE_DATA_BUFFER,
69734|     #define FSCTL_ENUM_USN_DATA
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 44, METHOD_NEITHER,
| FILE_ANY_ACCESS) // MFT_ENUM_DATA,
69735|     #define FSCTL_SECURITY_ID_CHECK
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 45, METHOD_NEITHER,
| FILE_READ_DATA) // BULK_SECURITY_TEST_DATA,
69736|     #define FSCTL_READ_USN_JOURNAL
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 46, METHOD_NEITHER,
| FILE_ANY_ACCESS) // READ_USN_JOURNAL_DATA, USN
69737|     #define FSCTL_SET_OBJECT_ID_EXTENDED
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 47, METHOD_BUFFERED,
| FILE_SPECIAL_ACCESS)
69738|     #define FSCTL_CREATE_OR_GET_OBJECT_ID
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 48, METHOD_BUFFERED,
| FILE_ANY_ACCESS) // FILE_OBJECTID_BUFFER
69739|     #define FSCTL_SET_SPARSE
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 49, METHOD_BUFFERED,
| FILE_SPECIAL_ACCESS)
69740|     #define FSCTL_SET_ZERO_DATA
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 50, METHOD_BUFFERED,
| FILE_WRITE_DATA) // FILE_ZERO_DATA_INFORMATION,
69741|     #define FSCTL_QUERY_ALLOCATED_RANGES
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 51, METHOD_NEITHER,
| FILE_READ_DATA) // FILE_ALLOCATED_RANGE_BUFFER,
| FILE_ALLOCATED_RANGE_BUFFER
69742|     #define FSCTL_ENABLE_UPGRADE
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 52, METHOD_BUFFERED,
| FILE_WRITE_DATA)
69743|     #define FSCTL_SET_ENCRYPTION
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 53, METHOD_NEITHER,
| FILE_ANY_ACCESS) // ENCRYPTION_BUFFER,
| DECRYPTION_STATUS_BUFFER
69744|     #define FSCTL_ENCRYPTION_FSCTL_IO
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 54, METHOD_NEITHER,
| FILE_ANY_ACCESS)
69745|     #define FSCTL_WRITE_RAW_ENCRYPTED
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 55, METHOD_NEITHER,
| FILE_SPECIAL_ACCESS) // ENCRYPTED_DATA_INFO,
69746|     #define FSCTL_READ_RAW_ENCRYPTED
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 56, METHOD_NEITHER,
| FILE_SPECIAL_ACCESS) // REQUEST_RAW_ENCRYPTED_DATA,
| ENCRYPTED_DATA_INFO
69747|     #define FSCTL_CREATE_USN_JOURNAL
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 57, METHOD_NEITHER,
| FILE_ANY_ACCESS) // CREATE_USN_JOURNAL_DATA,
69748|     #define FSCTL_READ_FILE_USN_DATA
| CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 58, METHOD_NEITHER,
| FILE_ANY_ACCESS) // Read the Usn Record for a file
69749|     #define FSCTL_WRITE_USN_CLOSE_RECORD

```

```

    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 59, METHOD_NEITHER,
    | FILE_ANY_ACCESS) // Generate Close Usn Record
69750|     #define FSCTL_EXTEND_VOLUME
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 60, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
69751|     #define FSCTL_QUERY_USN_JOURNAL
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 61, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
69752|     #define FSCTL_DELETE_USN_JOURNAL
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 62, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
69753|     #define FSCTL_MARK_HANDLE
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 63, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
69754|     #define FSCTL_SIS_COPYFILE
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 64, METHOD_BUFFERED,
    | FILE_ANY_ACCESS)
69755|     #define FSCTL_SIS_LINK_FILES
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 65, METHOD_BUFFERED,
    | FILE_READ_DATA | FILE_WRITE_DATA)
69756|     #define FSCTL_HSM_MSG
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 66, METHOD_BUFFERED,
    | FILE_READ_DATA | FILE_WRITE_DATA)
69757|     #define FSCTL_NSS_CONTROL
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 67, METHOD_BUFFERED,
    | FILE_WRITE_DATA)
69758|     #define FSCTL_HSM_DATA
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 68, METHOD_NEITHER,
    | FILE_READ_DATA | FILE_WRITE_DATA)
69759|     #define FSCTL_RECALL_FILE
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 69, METHOD_NEITHER,
    | FILE_ANY_ACCESS)
69760|     #define FSCTL_NSS_RCONTROL
    | CTL_CODE(FILE_DEVICE_FILE_SYSTEM, 70, METHOD_BUFFERED,
    | FILE_READ_DATA)
69761|     #endif /* _WIN32_WINNT >= 0x0500 */
69762| #endif
69763|
69764|
69765| //
69766| // Global storage for this file system filter driver.
69767| //
69768|
69769| LIST_ENTRY FsDeviceQueue;
69770| ERESOURCE FsLock;
69771|
69772| LIST_ENTRY
    | OpenSnapshotFiles={&OpenSnapshotFiles,&OpenSnapshotFiles
    | };
69773| LIST_ENTRY PSMFiles={&PSMFiles,&PSMFiles};

```

```

69774| KSPIN_LOCK ReadOnlySpinLock={0};
69775| ULONG NewFileWritesAllowed=0;
69776|
69777| char *File_GetFSCTLFunctionName( ULONG Minor, ULONG
    | FsCtl );
69778|
69779|
69780|
69781| BOOLEAN IsFileNameOneOfOurs( PUNICODE_STRING FileName )
69782| {
69783|     // FileName = "\Persistent Storage Manager
    | State\69a769a7007e00000000000000000000.diff.psm"
69784|     // we will return back true for any files in our
    | directory
69785|     WCHAR *p=PSM_PRIVATE_DIR_SLASH;
69786|     ULONG Size = wcslen(p);
69787|     if(Size<=FileName->Length) {
69788|         if(_wcsnicmp(FileName->Buffer+1,p,Size)==0) {
69789|             return TRUE;
69790|         } else {
69791|             return FALSE;
69792|         }
69793|     } else {
69794|         return FALSE;
69795|     }
69796| }
69797|
69798| BOOLEAN IsFileNameClusterFile( PUNICODE_STRING FileName
    | )
69799| {
69800|     // FileName = "\zCluster"
69801|     // we will return back true if the special cluster
    | file
69802|     WCHAR *p=L"zClusterOnlineChk.tmp";
69803|     ULONG Size = wcslen(p);
69804|     if(Size<=FileName->Length) {
69805|         if(_wcsnicmp(FileName->Buffer+1,p,Size)==0) {
69806|             return TRUE;
69807|         } else {
69808|             return FALSE;
69809|         }
69810|     } else {
69811|         return FALSE;
69812|     }
69813| }
69814|
69815| BOOLEAN IsFileNameRootDir( PUNICODE_STRING FileName )
69816| {
69817|     WCHAR *p=L"\\";
69818|     ULONG Size = wcslen(p);

```



```

69819|  if(Size<=FileName->Length) {
69820|      if(_wcsicmp(FileName->Buffer,p)==0) {
69821|          return TRUE;
69822|      } else {
69823|          return FALSE;
69824|      }
69825|  } else {
69826|      return FALSE;
69827|  }
69828| }
69829|
69830|
69831|
69832| NTSTATUS
69833| FilterDriverEntry(
69834|     IN PDRIVER_OBJECT DriverObject,
69835|     IN PUNICODE_STRING RegistryPath
69836| )
69837|
69838| /*++
69839|
69840| Routine Description:
69841|
69842|   This is the initialization routine for the general
69843|   | purpose file system
69844|   | filter driver. This routine creates the device
69845|   | object that represents this
69846|   | driver in the system and registers it for watching
69847|   | all file systems that
69848|   | register or unregister themselves as active file
69849|   | systems.
69850|
69851| Arguments:
69852|
69853|   DriverObject - Pointer to driver object created by
69854|   | the system.
69855|
69856| Return Value:
69857|
69858|   The function value is the final status from the
69859|   | initialization operation.
69860|
69861| --*/
69862| {
69863| #if DO_FILE_SYSTEM_FILTER
69864|     UNICODE_STRING nameString;
69865|     PDEVICE_OBJECT deviceObject;
69866|     PFILE_OBJECT fileObject;
69867|     NTSTATUS status;

```

```

69863| PFAST_IO_DISPATCH fastIoDispatch;
69864| ULONG i;
69865| PFS_OBJECT_EXTENSION deviceExtension;
69866|
69867|
69868| | Reg_GetULONGKey(RegistryPath,L"FileSysFilter",1,&i);
69869| // told not to install the file system filter
69870| if ( i==0 ) {
69871|     return STATUS_SUCCESS;
69872| }
69873|
69874| Debug(DEBUG_SFILTER,("SFILTER: Creating FileSystem
69875| | Object\n"));
69876| //
69877| // Create the device object.
69878| //
69879|
69880| RtlInitUnicodeString( &nameString,
69881| | L"\\FileSystem\\PSMFSFil" );
69882| #ifdef DEBUG_EXTENSION
69883|     ULONG SizeOfDeviceExt = sizeof(DEVICE_EXTENSION);
69884| #else
69885|     ULONG SizeOfDeviceExt = sizeof( FS_OBJECT_EXTENSION
69886| | );
69887| #endif
69888|
69889| status = IoCreateDevice(
69890|     DriverObject,
69891|     SizeOfDeviceExt,
69892|     &nameString,
69893|     | FILE_DEVICE_DISK_FILE_SYSTEM,
69894|     0,
69895|     FALSE,
69896|     &deviceObject
69897| );
69898| if ( !NT_SUCCESS( status ) ) {
69899|     Debug(DEBUG_SFILTER,("SFILTER: Error creating
69900| | FileSystem Object, error: %08x\n", status ));
69901|     return status;
69902| }
69903| //
69904| // Initialize the driver object with this device
69905| | driver's entry points.
69906| //
69907| // only do FS specific ones

```

```

69906|
| DriverObject->MajorFunction[IRP_MJ_FILE_SYSTEM_CONTROL]
| = SfFsControl;
69907|
| DriverObject->MajorFunction[IRP_MJ_QUERY_INFORMATION] =
| SfFsQueryInformation;
69908| DriverObject->MajorFunction[IRP_MJ_SET_INFORMATION]
| = SfFsSetInformation;
69909|
| DriverObject->MajorFunction[IRP_MJ_DIRECTORY_CONTROL] =
| SfFsDirectoryControl;
69910|
69911| #if 0
69912| [00] IRP_MJ_CREATE                bfe43e90
| psman5!PSManCreate
69913| [01] IRP_MJ_CREATE_NAMED_PIPE    bfe902c5
| psman5!PSManFSPassThru
69914| [02] IRP_MJ_CLOSE                bfe43840
| psman5!PSManClose
69915| [03] IRP_MJ_READ                 bfe86ae0
| psman5!PSManRead
69916| [04] IRP_MJ_WRITE                bfe9efb0
| psman5!PSManWrite
69917| [05] IRP_MJ_QUERY_INFORMATION    bfe96280
| psman5!SfFsQueryInformation
69918| [06] IRP_MJ_SET_INFORMATION        bfe967c3
| psman5!SfFsSetInformation
69919| [07] IRP_MJ_QUERY_EA              bfe902c5
| psman5!PSManFSPassThru
69920| [08] IRP_MJ_SET_EA               bfe902c5
| psman5!PSManFSPassThru
69921| [09] IRP_MJ_FLUSH_BUFFERS        bfe6b0d0
| psman5!PSManFlush
69922| [0a] IRP_MJ_QUERY_VOLUME_INFORMATION bfe902c5
| psman5!PSManFSPassThru
69923| [0b] IRP_MJ_SET_VOLUME_INFORMATION bfe902c5
| psman5!PSManFSPassThru
69924| [0c] IRP_MJ_DIRECTORY_CONTROL    bfe96ba0
| psman5!SfFsDirectoryControl
69925| [0d] IRP_MJ_FILE_SYSTEM_CONTROL  bfe916fd
| psman5!SfFsControl
69926| [0e] IRP_MJ_DEVICE_CONTROL        bfe5160a
| psman5!PSManDeviceControl
69927| [0f] IRP_MJ_INTERNAL_DEVICE_CONTROL bfe902c5
| psman5!PSManFSPassThru
69928| [10] IRP_MJ_SHUTDOWN              bfe97dbc
| psman5!PSManShutdown
69929| [11] IRP_MJ_LOCK_CONTROL          bfe902c5
| psman5!PSManFSPassThru
69930| [12] IRP_MJ_CLEANUP              bfe432d0

```

```

    | psman5!PSManCleanup
69931| [13] IRP_MJ_CREATE_MAILSLOT          bfe902c5
    | psman5!PSManFSPassThru
69932| [14] IRP_MJ_QUERY_SECURITY           bfe902c5
    | psman5!PSManFSPassThru
69933| [15] IRP_MJ_SET_SECURITY             bfe902c5
    | psman5!PSManFSPassThru
69934| [16] IRP_MJ_POWER                   bfe84a60
    | psman5!PSManPower
69935| [17] IRP_MJ_SYSTEM_CONTROL           bfe902c5
    | psman5!PSManFSPassThru
69936| [18] IRP_MJ_DEVICE_CHANGE           bfe902c5
    | psman5!PSManFSPassThru
69937| [19] IRP_MJ_QUERY_QUOTA            bfe902c5
    | psman5!PSManFSPassThru
69938| [1a] IRP_MJ_SET_QUOTA               bfe902c5
    | psman5!PSManFSPassThru
69939| [1b] IRP_MJ_PNP                    bfe83280
    | psman5!PSManPnp
69940| #endif
69941|
69942|
69943| //
69944| // Allocate fast I/O data structure and fill it in.
69945| //
69946|
69947| fastIoDispatch =
    | (PFAST_IO_DISPATCH)MemAllocatePoolWithTag(
    | NonPagedPool, sizeof( FAST_IO_DISPATCH
    | ),PSM_DISPATCH_TABLE_TAG );
69948| if ( !fastIoDispatch ) {
69949|     Debug(DEBUG_SFILTER,("SFILTER: Out of memory
    | for Dispatch table\n"));
69950|     IoDeleteDevice( deviceObject );
69951|     return STATUS_INSUFFICIENT_RESOURCES;
69952| }
69953|
69954| RtlZeroMemory( fastIoDispatch, sizeof(
    | FAST_IO_DISPATCH ) );
69955| fastIoDispatch->SizeOfFastIoDispatch = sizeof(
    | FAST_IO_DISPATCH );
69956| fastIoDispatch->FastIoCheckIfPossible =
    | SfFastIoCheckIfPossible;
69957| fastIoDispatch->FastIoRead = SfFastIoRead;
69958| fastIoDispatch->FastIoWrite = SfFastIoWrite;
69959| fastIoDispatch->FastIoQueryBasicInfo =
    | SfFastIoQueryBasicInfo;
69960| fastIoDispatch->FastIoQueryStandardInfo =
    | SfFastIoQueryStandardInfo;
69961| fastIoDispatch->FastIoLock = SfFastIoLock;

```

```

69962| fastIoDispatch->FastIoUnlockSingle =
| SfFastIoUnlockSingle;
69963| fastIoDispatch->FastIoUnlockAll =
| SfFastIoUnlockAll;
69964| fastIoDispatch->FastIoUnlockAllByKey =
| SfFastIoUnlockAllByKey;
69965| fastIoDispatch->FastIoDeviceControl =
| SfFastIoDeviceControl;
69966| fastIoDispatch->FastIoDetachDevice =
| SfFastIoDetachDevice;
69967| fastIoDispatch->FastIoQueryNetworkOpenInfo =
| SfFastIoQueryNetworkOpenInfo;
69968| fastIoDispatch->MdlRead = SfFastIoMdlRead;
69969| fastIoDispatch->MdlReadComplete =
| SfFastIoMdlReadComplete;
69970| fastIoDispatch->PrepareMdlWrite =
| SfFastIoPrepareMdlWrite;
69971| fastIoDispatch->MdlWriteComplete =
| SfFastIoMdlWriteComplete;
69972| fastIoDispatch->FastIoReadCompressed =
| SfFastIoReadCompressed;
69973| fastIoDispatch->FastIoWriteCompressed =
| SfFastIoWriteCompressed;
69974| fastIoDispatch->MdlReadCompleteCompressed =
| SfFastIoMdlReadCompleteCompressed;
69975| fastIoDispatch->MdlWriteCompleteCompressed =
| SfFastIoMdlWriteCompleteCompressed;
69976| fastIoDispatch->FastIoQueryOpen =
| SfFastIoQueryOpen;
69977|
69978| // no way to get the device object so we can send
| it down.
69979| // The filter drivers i am looking at dont filter
| this, or do it wrong.
69980| // fastIoDispatch->AcquireFileForNtCreateSection =
| SfFastIoAcquireFileForNtCreateSection;
69981| // fastIoDispatch->ReleaseFileForNtCreateSection =
| SfFastIoReleaseFileForNtCreateSection;
69982| fastIoDispatch->AcquireForModWrite =
| SfFastIoAcquireForModWrite;
69983| fastIoDispatch->ReleaseForModWrite =
| SfFastIoReleaseForModWrite;
69984| fastIoDispatch->AcquireForCcFlush =
| SfFastIoAcquireForCcFlush;
69985| fastIoDispatch->ReleaseForCcFlush =
| SfFastIoReleaseForCcFlush;
69986|
69987| KeInitializeSpinLock(&ReadOnlySpinLock);
69988| InitializeListHead(&OpenSnapshotFiles);
69989|

```

```

69990| //
69991| // Note that there is no safe way to filter
    | AcquireFileForNtCreateSection
69992| // and ReleaseFileForNtCreateSection.
69993| //
69994|
69995| DriverObject->FastIoDispatch = fastIoDispatch;
69996|
69997| //
69998| // Initialize global data structures.
69999| //
70000|
70001| InitializeListHead( &FsDeviceQueue );
70002| ExInitializeResource( &FsLock );
70003|
70004| //
70005| // Register this driver for watching file systems
    | coming and going.
70006| //
70007|
70008| status = IoRegisterFsRegistrationChange(
    | DriverObject, SsFsNotification );
70009| if ( !NT_SUCCESS( status ) ) {
70010|     Debug(DEBUG_SFILTER,( "SFILTER: Error
    | registering FS change notification, error: %08x\n",
    | status ));
70011|     ExDeleteResource( &FsLock );
70012|     IoDeleteDevice( deviceObject );
70013|     return status;
70014| }
70015|
70016| //
70017| // Indicate that the type for this device object is
    | a primary, not a filter
70018| // device object so that it doesn't accidentally
    | get used to call a file
70019| // system.
70020| //
70021|
70022| #ifdef DEBUG_EXTENSION
70023|
    | ((PDEVICE_EXTENSION)(deviceObject->DeviceExtension))->Ob
    | jectType = OBJECT_FS_OBJECT;
70024|
    | ((PDEVICE_EXTENSION)(deviceObject->DeviceExtension))->Re
    | alDeviceExtension =
    | MemAllocatePoolWithTag(NonPagedPool,FS_OBJECT_EXTENSION_
    | SIZE,DEVEXTTAG);
70025|
    | RtlZeroMemory(((PDEVICE_EXTENSION)(deviceObject->DeviceE

```

```

    | xtension))->RealDeviceExtension,FS_OBJECT_EXTENSION_SIZE
    | );
70026| #endif
70027|
70028|     deviceExtension =
    | (PFS_OBJECT_EXTENSION)GetDeviceExtension(deviceObject);
70029|     deviceExtension->DeviceObject = deviceObject;
70030|     deviceExtension->DriverObject = DriverObject;
70031|     deviceExtension->ObjectType = OBJECT_FS_OBJECT;
70032|
70033|     //
70034|     // Attempt to open the RAW device object since this
    | driver has already
70035|     // been started by system initialization and will
    | not make it through
70036|     // the normal file system notification procedures
    | otherwise.
70037|     //
70038|
70039|     RtlInitUnicodeString( &nameString,
    | L"\\Device\\RawDisk" );
70040|     status = IoGetDeviceObjectPointer(
70041|                                     &nameString,
70042|
    | FILE_READ_ATTRIBUTES,
70043|                                     &fileObject,
70044|                                     &deviceObject
70045|                                     );
70046|
70047|     if ( NT_SUCCESS( status ) ) {
70048|         Debug(DEBUG_SFILTER,("SFILTER: Attaching to RAW
    | FileSystem\n"));
70049|         SfsFsNotification( deviceObject, TRUE );
70050|         ObDereferenceObject( fileObject );
70051|     }
70052| #endif
70053|     return STATUS_SUCCESS;
70054| }
70055|
70056| /*-----
    | -----*/
70057| NTSTATUS
70058| PManFSPassThru(
70059|     IN PDEVICE_OBJECT DeviceObject,
70060|     IN PIRP Irp
70061| )
70062|
70063| /*++
70064|
70065| Routine Description:

```

```

70066|     Passthru function that doesnt generate an
| outstanding requests.. use
70067|     this function instead of PSMPassThru to prevent
| miscounts.
70068|
70069| Arguments:
70070|
70071|     DeviceObject
70072|     Irp
70073|
70074| Return Value:
70075|
70076|     NTSTATUS
70077|
70078| --*/
70079|
70080| {
70081|     if (
| (PsmGetObjectType(DeviceObject)!=OBJECT_FS_FILTER) &&
| (PsmGetObjectType(DeviceObject)!=OBJECT_FILTEREDDISK) )
| {
70082|         Debug(DEBUG_PASSTHRU | DEBUG_ERROR,("Error!
| Device is not filter device, Failing request\n"));
70083|
70084|         Irp->IoStatus.Status =
| STATUS_INVALID_DEVICE_REQUEST;
70085|         Irp->IoStatus.Information = 0;
70086|         IoCompleteRequest(Irp, IO_NO_INCREMENT);
70087|         return STATUS_INVALID_DEVICE_REQUEST;
70088|     }
70089|
70090|     IoSkipCurrentIrpStackLocation( Irp );
70091|
70092|     if (
| PsmGetObjectType(DeviceObject)==OBJECT_FILTEREDDISK ) {
70093|         return
| IoCallDriver(((PFILTERED_EXTENSION)GetDeviceExtension(De
| viceObject))->TargetDeviceObject, Irp);
70094|     } else {
70095|         return
| IoCallDriver(((PFS_FILTER_EXTENSION)GetDeviceExtension(D
| eviceObject))->TargetDeviceObject, Irp);
70096|     }
70097| } // end PSManFSPassThru()
70098|
70099|
70100| BOOLEAN FileIsOpenOnSnapShot ( PFILE_OBJECT FileObject,
| BOOLEAN Remove )
70101| {
70102|     KIRQL oldIrql;

```



```

70103|  PLIST_ENTRY ListEntry;
70104|  tOpenSnapShotFiles *p;
70105|  BOOLEAN Found=0;
70106|
70107|  pmAcquireSpinLock(&ReadOnlySpinLock,&oldIrql);
70108|  ListEntry = OpenSnapShotFiles.Flink;
70109|
70110|  while ( ListEntry! =&OpenSnapShotFiles ) {
70111|      p =
          | CONTAINING_RECORD(ListEntry,tOpenSnapShotFiles,ListEntry
          | );
70112|
70113|      if ( p->FileObject==FileObject ) {
70114|          if ( Remove ) {
70115|              RemoveEntryList(&p->ListEntry);
70116|              MemFreePool(p);
70117|          }
70118|          Found=1;
70119|          break;
70120|      }
70121|      ListEntry = ListEntry->Flink;
70122|  }
70123|
70124|  pmReleaseSpinLock(&ReadOnlySpinLock,oldIrql);
70125|  return Found;
70126| }
70127|
70128|
70129| BOOLEAN FileIsReadOnly ( PFILE_OBJECT FileObject )
70130| {
70131|     KIRQL oldIrql;
70132|     PLIST_ENTRY ListEntry;
70133|     tOpenSnapShotFiles *p;
70134|     BOOLEAN ReadOnly = FALSE;
70135|
70136|     pmAcquireSpinLock(&ReadOnlySpinLock,&oldIrql);
70137|     ListEntry = OpenSnapShotFiles.Flink;
70138|
70139|     while ( ListEntry! =&OpenSnapShotFiles ) {
70140|         p =
            | CONTAINING_RECORD(ListEntry,tOpenSnapShotFiles,ListEntry
            | );
70141|
70142|         if ( p->FileObject==FileObject ) {
70143|             ReadOnly = p->ReadOnly;
70144|             break;
70145|         }
70146|         ListEntry = ListEntry->Flink;
70147|     }
70148|

```

```

70149| pmReleaseSpinLock(&ReadOnlySpinLock,oldIrql);
70150| return ReadOnly;
70151| }
70152|
70153|
70154| BOOLEAN FileIsPSM( PFILE_OBJECT FileObject, BOOLEAN
    | Remove )
70155| {
70156|     KIRQL oldIrql;
70157|     PLIST_ENTRY ListEntry;
70158|     tOpenSnapShotFiles *p;
70159|     BOOLEAN Found=0;
70160|
70161|     pmAcquireSpinLock(&ReadOnlySpinLock,&oldIrql);
70162|     ListEntry = PSMFiles.Flink;
70163|
70164|     while ( ListEntry!=&PSMFiles ) {
70165|         p =
    | CONTAINING_RECORD(ListEntry,tOpenSnapShotFiles,ListEntry
    | );
70166|
70167|         if ( p->FileObject==FileObject ) {
70168|             if ( Remove ) {
70169|                 RemoveEntryList(&p->ListEntry);
70170|                 MemFreePool(p);
70171|             }
70172|             Found=1;
70173|             break;
70174|         }
70175|         ListEntry = ListEntry->Flink;
70176|     }
70177|
70178|     pmReleaseSpinLock(&ReadOnlySpinLock,oldIrql);
70179|     return Found;
70180| }
70181|
70182|
70183| NTSTATUS
70184| PSManFSClose(
70185|     IN PDEVICE_OBJECT DeviceObject,
70186|     IN PIRP Irp
70187| )
70188| {
70189|     PIO_STACK_LOCATION IrpSp =
    | IoGetCurrentIrpStackLocation( Irp );
70190|
70191|     // cleanup list
70192|     FileIsOpenOnSnapShot(IrpSp->FileObject,TRUE);
70193|     FileIsPSM(IrpSp->FileObject,TRUE);
70194|

```

```

70195| return PSMANFSPassThru(DeviceObject,Irp);
70196| }
70197|
70198| #ifdef DEBUG
70199| char *GetCreateDispositionString( ULONG Options )
70200| {
70201|     ULONG CreateDisposition = (Options >> 24) &
70202|         | 0x000000ff;
70203|     switch(CreateDisposition) {
70204|         case FILE_SUPERSEDE : return "Supercede";
70205|         case FILE_OPEN      : return "Open";
70206|         case FILE_CREATE    : return "Create";
70207|         case FILE_OPEN_IF   : return "OpenIf";
70208|         case FILE_OVERWRITE : return "Overwrite";
70209|         case FILE_OVERWRITE_IF : return "OverwriteIf";
70210|         default:
70211|             return "Unknown";
70212|     }
70213| }
70214| char *GetOptionsString( ULONG Options, char *Buffer )
70215| {
70216|     strcpy(Buffer,"");
70217|
70218|     if(Options & 0x00000001)
70219|         strcat(Buffer,"DIRECTORY,");
70220|     if(Options & 0x00000002)
70221|         strcat(Buffer,"WRITE_THROUGH,");
70222|     if(Options & 0x00000004)
70223|         strcat(Buffer,"SEQUENTIAL_ONLY,");
70224|     if(Options & 0x00000008)
70225|         strcat(Buffer,"NO_BUFFERING,");
70226|     if(Options & 0x00000010)
70227|         strcat(Buffer,"SYNC_ALERT,");
70228|     if(Options & 0x00000020)
70229|         strcat(Buffer,"SYNC_NONALERT,");
70230|     if(Options & 0x00000040)
70231|         strcat(Buffer,"NON_DIRECTORY,");
70232|     if(Options & 0x00000080)
70233|         strcat(Buffer,"CREATE_TREE_CONNECTION,");
70234|     if(Options & 0x00000100)
70235|         strcat(Buffer,"COMPLETE_IF_OPLOCKED,");
70236|     if(Options & 0x00000200)
70237|         strcat(Buffer,"NO_EA_KNOWLEDGE,");
70238|     if(Options & 0x00000400)
70239|         strcat(Buffer,"OPEN_FOR_RECOVERY,");
70240|     if(Options & 0x00000800)
70241|         strcat(Buffer,"RANDOM_ACCESS,");
70242|     if(Options & 0x00001000)
70243|         strcat(Buffer,"DELETE_ON_CLOSE,");

```

```

70244|  if(Options & 0x00002000)
70245|      strcat(Buffer,"OPEN_BY_FILE_ID,");
70246|  if(Options & 0x00004000)
70247|      strcat(Buffer,"OPEN_FOR_BACKUP,");
70248|  if(Options & 0x00008000)
70249|      strcat(Buffer,"NO_COMPRESSION,");
70250|  if(Options & 0x00010000)
70251|      strcat(Buffer,"00010000,");
70252|  if(Options & 0x00020000)
70253|      strcat(Buffer,"00020000,");
70254|  if(Options & 0x00040000)
70255|      strcat(Buffer,"00040000,");
70256|  if(Options & 0x00080000)
70257|      strcat(Buffer,"00080000,");
70258|  if(Options & 0x00100000)
70259|      strcat(Buffer,"RESERVE_OPFILTER,");
70260|  if(Options & 0x00200000)
70261|      strcat(Buffer,"OPEN_REPARSE_POINT,");
70262|  if(Options & 0x00400000)
70263|      strcat(Buffer,"OPEN_NO_RECALL,");
70264|  if(Options & 0x00800000)
70265|      strcat(Buffer,"OPEN_FOR_FREE_SPACE_QUERY,");
70266|
70267|  return Buffer;
70268| }
70269|
70270| char *GetFileInfoClassString( FILE_INFORMATION_CLASS
    | Info )
70271| {
70272|  switch(Info) {
70273|      case FileDirectoryInformation : return
    | "FileDirectoryInformation"; // 1
70274|      case FileFullDirectoryInformation : return
    | "FileFullDirectoryInformation"; // 2
70275|      case FileBothDirectoryInformation : return
    | "FileBothDirectoryInformation"; // 3
70276|      case FileBasicInformation : return
    | "FileBasicInformation"; // 4 wdm
70277|      case FileStandardInformation : return
    | "FileStandardInformation"; // 5 wdm
70278|      case FileInternalInformation : return
    | "FileInternalInformation"; // 6
70279|      case FileEaInformation : return
    | "FileEaInformation"; // 7
70280|      case FileAccessInformation : return
    | "FileAccessInformation"; // 8
70281|      case FileNameInformation : return
    | "FileNameInformation"; // 9
70282|      case FileRenameInformation : return
    | "FileRenameInformation"; // 10

```

70283| case FileLinkInformation : return  
| "FileLinkInformation"; // 11  
70284| case FileNamesInformation : return  
| "FileNamesInformation"; // 12  
70285| case FileDispositionInformation : return  
| "FileDispositionInformation"; // 13  
70286| case FilePositionInformation : return  
| "FilePositionInformation"; // 14 wdm  
70287| case FileFullEaInformation : return  
| "FileFullEaInformation"; // 15  
70288| case FileModelInformation : return  
| "FileModelInformation"; // 16  
70289| case FileAlignmentInformation : return  
| "FileAlignmentInformation"; // 17  
70290| case FileAllInformation : return  
| "FileAllInformation"; // 18  
70291| case FileAllocationInformation : return  
| "FileAllocationInformation"; // 19  
70292| case FileEndOfFileInformation : return  
| "FileEndOfFileInformation"; // 20 wdm  
70293| case FileAlternateNameInformation : return  
| "FileAlternateNameInformation"; // 21  
70294| case FileStreamInformation : return  
| "FileStreamInformation"; // 22  
70295| case FilePipeInformation : return  
| "FilePipeInformation"; // 23  
70296| case FilePipeLocalInformation : return  
| "FilePipeLocalInformation"; // 24  
70297| case FilePipeRemoteInformation : return  
| "FilePipeRemoteInformation"; // 25  
70298| case FileMailslotQueryInformation : return  
| "FileMailslotQueryInformation"; // 26  
70299| case FileMailslotSetInformation : return  
| "FileMailslotSetInformation"; // 27  
70300| case FileCompressionInformation : return  
| "FileCompressionInformation"; // 28  
70301| case FileObjectIdInformation : return  
| "FileObjectIdInformation"; // 29  
70302| case FileCompletionInformation : return  
| "FileCompletionInformation"; // 30  
70303| case FileMoveClusterInformation : return  
| "FileMoveClusterInformation"; // 31  
70304| case FileQuotaInformation : return  
| "FileQuotaInformation"; // 32  
70305| case FileReparsePointInformation : return  
| "FileReparsePointInformation"; // 33  
70306| case FileNetworkOpenInformation : return  
| "FileNetworkOpenInformation"; // 34  
70307| case FileAttributeTagInformation : return  
| "FileAttributeTagInformation"; // 35

```

70308|     case FileTrackingInformation : return
70309|         | "FileTrackingInformation";    // 36
70310|     default:
70311|         return "unknown";
70312|     }
70313| }
70314| STATIC void DumpCreateInfo( PIRP Irp )
70315| {
70316|     UNICODE_STRING Empty;
70317|     char Buffer[256];
70318|     PIO_STACK_LOCATION IrpSp =
70319|         | IoGetCurrentIrpStackLocation( Irp );
70320|     #if 0
70321|         // taken from fastfat to show how to get
70322|         | different options
70323|         FileObject      = IrpSp->FileObject;
70324|         FileName        = FileObject->FileName;
70325|         RelatedFileObject =
70326|         | FileObject->RelatedFileObject;
70327|         AllocationSize  =
70328|         | Irp->Overlay.AllocationSize.LowPart;
70329|         EaBuffer         =
70330|         | Irp->AssociatedIrp.SystemBuffer;
70331|         DesiredAccess    =
70332|         | &IrpSp->Parameters.Create.SecurityContext->DesiredAccess
70333|         | ;
70334|         Options          =
70335|         | IrpSp->Parameters.Create.Options;
70336|         FileAttributes   =
70337|         | (UCHAR)(IrpSp->Parameters.Create.FileAttributes &
70338|         | ~FILE_ATTRIBUTE_NORMAL);
70339|         ShareAccess      =
70340|         | IrpSp->Parameters.Create.ShareAccess;
70341|         EaLength         =
70342|         | IrpSp->Parameters.Create.EaLength;
70343|         CreateDisposition = (Options >> 24) &
70344|         | 0x000000ff;
70345|     #endif
70346|     RtlInitUnicodeString(&Empty,L"<No name
70347|         | available>");
70348|     Debug(DEBUG_SFILTER,("SFILTER: %s '%wZ', co=%s\n",
70349|         | GetCreateDispositionString(IrpSp->Parameters.Create.Opti
70350|         | ons),
70351|         | IrpSp->FileObject->FileName.Length ?

```

```

    | &IrpSp->FileObject->FileName : &Empty,
70340|
    | GetOptionsString(IrpSp->Parameters.Create.Options,Buffer
    | ));
70341|
70342| }
70343| #endif
70344|
70345|
70346|
70347| STATIC
70348| NTSTATUS
70349| SfCreate(
70350|     IN PDEVICE_OBJECT DeviceObject,
70351|     IN PIRP Irp
70352| )
70353|
70354| /*++
70355|
70356| Routine Description:
70357|
70358| This function filters create/open operations. It
    | simply establishes an
70359| I/O completion routine to be invoked if the
    | operation was successful.
70360|
70361| Arguments:
70362|
70363| DeviceObject - Pointer to the target device object
    | of the create/open.
70364|
70365| Irp - Pointer to the I/O Request Packet that
    | represents the operation.
70366|
70367| Return Value:
70368|
70369| The function value is the status of the call to the
    | file system's entry
70370| point.
70371|
70372| --*/
70373|
70374| {
70375| #if DO_FILE_SYSTEM_FILTER
70376|
70377| PAGED_CODE();
70378|
70379| //Debug(DEBUG_SFILTER,("SFILTER: Create called
    | %08x, %08x\n",DeviceObject,Irp));
70380|

```

```

70381|   if (
      | PsmGetObjectType(DeviceObject)==OBJECT_FS_OBJECT ) {
70382|       Debug(DEBUG_SFILTER,"SFILTER: FileSystem
      | object and not the FS filter\n");
70383|       Irp->IoStatus.Status = STATUS_SUCCESS;
70384|       Irp->IoStatus.Information = FILE_OPENED;
70385|
70386|       IoCompleteRequest( Irp, IO_NO_INCREMENT );
70387|       return STATUS_SUCCESS;
70388|   }
70389|
70390|
      | ASSERT(PsmGetObjectType(DeviceObject)==OBJECT_FS_FILTER)
      | ;
70391|
70392|   PIO_STACK_LOCATION IrpSp =
      | IoGetCurrentIrpStackLocation( Irp );
70393|
70394|   if (
      | (((PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceObject)
      | )->Virtual) &&
70395|
      | (((PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceObject)
      | )->PSMStorageObject) ) {
70396|       PFS_FILTER_EXTENSION
      | deviceExtension=(PFS_FILTER_EXTENSION)GetDeviceExtension
      | (DeviceObject);
70397|       pOpenSnapShotFiles Context=NULL;
70398|       PVDISK_EXTENSION
      | VDiskExt=(PVDISK_EXTENSION)GetDeviceExtension(deviceExte
      | nsion->PSMStorageObject);
70399|
70400|       // see if extended handling is off
70401|       if(!(gVDiskIOHandling &
      | PSM_VDISK_FLAG_ALLOW_PSM_FILE_OPEN)) {
70402|           if(IrpSp->FileObject->FileName.Length) {
70403|               // FIXFIXFIX need to also check
      | relative opens.
70404|
      | if(IsFileNameOneOfOurs(&IrpSp->FileObject->FileName)) {
70405|               // never allow an open on the
      | virtual volume
70406|               Irp->IoStatus.Status =
      | STATUS_ACCESS_DENIED;
70407|               Irp->IoStatus.Information = 0;
70408|
70409|               IoCompleteRequest( Irp,
      | IO_NO_INCREMENT );
70410|               return STATUS_ACCESS_DENIED;
70411|           }

```



```

70412|     }
70413| }
70414|
70415|     ULONG SnapShotIsReadOnly = FALSE;
70416|     if ( (VDiskExt->SnapShot) &&
        | pPersistentDictionary(VDiskExt->SnapShot->Dictionary)->I
        | sReadOnly() ) {
70417|         // virtual volume is marked readonly
70418|         SnapShotIsReadOnly = TRUE;
70419| #ifdef DEBUG
70420|     #if DO_ALL_SFILTER
70421|         UNICODE_STRING Empty;
70422|
70423|         RtlInitUnicodeString(&Empty,L"No name
        | available");
70424|         Debug(DEBUG_SFILTER,("SFILTER: Create %08x
        | '%wZ', co=%08x, da=%08x, sa=%08x, fa=%08x\n",
70425|             IrpSp->FileObject,
70426|             | IrpSp->FileObject->FileName.Length ?
        | &IrpSp->FileObject->FileName : &Empty,
70427|             | IrpSp->Parameters.Create.Options,
70428|             | IrpSp->Parameters.Create.SecurityContext->DesiredAccess,
70429|             | IrpSp->Parameters.Create.ShareAccess,
70430|             | IrpSp->Parameters.Create.FileAttributes));
70431|     #endif /*DO_ALL_SFILTER*/
70432| #endif /*DEBUG*/
70433|         // check to see if trying to create a new
        | file
70434|
70435|         ASSERT(
        | IrpSp->Parameters.Create.SecurityContext != NULL );
70436|
70437| #if 0
70438|         // taken from fastfat to show how to get
        | different options
70439|         FileObject      = IrpSp->FileObject;
70440|         FileName        = FileObject->FileName;
70441|         RelatedFileObject =
        | FileObject->RelatedFileObject;
70442|         AllocationSize  =
        | Irp->Overlay.AllocationSize.LowPart;
70443|         EaBuffer        =
        | Irp->AssociatedIrp.SystemBuffer;
70444|         DesiredAccess    =
        | &IrpSp->Parameters.Create.SecurityContext->DesiredAccess

```

```

| ;
70445|         Options      =
| IrpSp->Parameters.Create.Options;
70446|         FileAttributes =
| (UCHAR)(IrpSp->Parameters.Create.FileAttributes &
| ~FILE_ATTRIBUTE_NORMAL);
70447|         ShareAccess    =
| IrpSp->Parameters.Create.ShareAccess;
70448|         EaLength       =
| IrpSp->Parameters.Create.EaLength;
70449|         CreateDisposition = (Options >> 24) &
| 0x000000ff;
70450| #endif
70451|
70452|         // we allow writes when this is set.
70453|         // this is so we can delete the snapshot
| links
70454|         if ( !NewFileWritesAllowed ) {
70455|             ULONG CreateDisposition =
| (IrpSp->Parameters.Create.Options >> 24) & 0x000000ff;
70456|
70457|             if (
70458|                 (CreateDisposition == FILE_CREATE)
| ||
70459|                 (CreateDisposition ==
| FILE_SUPERSEDE) ||
70460|                 (CreateDisposition ==
| FILE_OVERWRITE) ||
70461|                 (CreateDisposition ==
| FILE_OVERWRITE_IF)
70462|             ) {
70463|                 Debug(DEBUG_SFILTER,("SFILTER:
| Create specified new file or overwrite!\n"));
70464|                 WriteProtected:
70465|                 Irp->IoStatus.Status =
| STATUS_MEDIA_WRITE_PROTECTED;
70466|                 Irp->IoStatus.Information = 0;
70467|
70468|                 IoCompleteRequest( Irp,
| IO_NO_INCREMENT );
70469|                 return
| STATUS_MEDIA_WRITE_PROTECTED;
70470|             } // attempt to create new file
70471|
70472|             if ( (CreateDisposition ==
| FILE_OPEN_IF) ) {
70473|                 // if told to create if not there,
| change it to only open. Our completion
70474|                 // routine will handle the case
| where it doesnt exist

```

```

70475|           IrpSp->Parameters.Create.Options &=
| ~(FILE_OPEN_IF << 24);
70476|           IrpSp->Parameters.Create.Options |=
| (FILE_OPEN << 24);
70477|       }
70478|       // dont allow deletes
70479|       if ( IrpSp->Parameters.Create.Options &
| FILE_DELETE_ON_CLOSE ) {
70480|           Debug(DEBUG_SFILTER,("SFILTER:
| Create: deletion attempted\n"));
70481|           goto WriteProtected;
70482|       }
70483|
70484|       // see if extended checks are disabled
70485|       if ( !(gVDiskIOHandling &
| PSM_VDISK_FLAG_ALLOW_OPEN_FOR_WRITE) ) {
70486|           // attempted open with write access
70487|           if (
70488|
| (IrpSp->Parameters.Create.SecurityContext->DesiredAccess
| & GENERIC_WRITE) ||
70489|
| (IrpSp->Parameters.Create.SecurityContext->DesiredAccess
| & FILE_WRITE_DATA) ||
70490|
| (IrpSp->Parameters.Create.SecurityContext->DesiredAccess
| & FILE_WRITE_ATTRIBUTES) ||
70491|
| (IrpSp->Parameters.Create.SecurityContext->DesiredAccess
| & FILE_APPEND_DATA) ||
70492|
| (IrpSp->Parameters.Create.SecurityContext->DesiredAccess
| & FILE_WRITE_EA)
70493|       ) {
70494|           Debug(DEBUG_SFILTER,("SFILTER:
| Create: create attempted with write access\n"));
70495|           goto WriteProtected;
70496|       }
70497|       } // extended checks
70498|       } // not adding drives
70499|       } // virtual is readonly
70500|
70501|       | Context=(pOpenSnapShotFiles)MemAllocatePoolWithTag(NonPa
| gedPool,sizeof(tOpenSnapShotFiles),PSM_READONLY_FILE_TAG
| );
70502|
70503|       // add object onto list of valid creates
70504|       if ( Context ) {
70505|           RtlZeroMemory (Context,

```

```

    | sizeof(tOpenSnapShotFiles));
70506|     Context->FileObject = IrpSp->FileObject;
70507|     Context->ReadOnly = TRUE;
70508|     if ( NewFileWritesAllowed ||
    | !SnapShotIsReadOnly ) {
70509|         Context->ReadOnly = FALSE;
70510|     }
70511|
    | ExInterlockedInsertTailList(&OpenSnapShotFiles,&Context-
    | >ListEntry,&ReadOnlySpinLock);
70512|     }
70513|
70514|     //
70515|     // Get a pointer to the current stack location
    | in the IRP. This is where
70516|     // the function codes and parameters are
    | stored.
70517|     //
70518|
70519|     IrpSp = IoGetCurrentIrpStackLocation( Irp );
70520|
70521|     //
70522|     // If debugging is enabled, do the processing
    | required to see the packet
70523|     // upon its completion. Otherwise, let the
    | request go w/no further
70524|     // processing.
70525|     //
70526|
70527|     IoCopyCurrentIrpStackLocationToNext( Irp );
70528|
70529|     IoSetCompletionRoutine(
70530|         Irp,
70531|         SfCreateCompletion,
70532|         Context,
70533|         TRUE,
70534|         TRUE,
70535|         TRUE
70536|     );
70537|
70538|     //
70539|     // Now call the appropriate file system driver
    | with the request.
70540|     //
70541|
70542|     return IoCallDriver(
    | deviceExtension->TargetDeviceObject, Irp );
70543| } // virtual
70544|
70545| // see if extended handling is off

```

```

70546|    if(!(gVDiskIOHandling &
| PSM_VDISK_FLAG_ALLOW_PSM_FILE_OPEN)) {
70547|        if(IrpSp->FileObject->FileName.Length) {
70548|            // FIXFIXFIX need to also check relative
| opens.
70549|
| if(IsFileNameOneOfOurs(&IrpSp->FileObject->FileName)) {
70550|            tOpenSnapShotFiles
| *Context=(pOpenSnapShotFiles)MemAllocatePoolWithTag(NonP
| agedPool,sizeof(tOpenSnapShotFiles),PSM_READONLY_FILE_TA
| G);
70551|
70552|            // add object onto list of valid
| creates
70553|            if ( Context ) {
70554|                Context->FileObject =
| IrpSp->FileObject;
70555|                Debug(DEBUG_SFILTER,("SFILTER: PSM
| file %08x
| '%wZ'\n",Context->FileObject,&IrpSp->FileObject->FileNam
| e));
70556|
| ExInterlockedInsertTailList(&PSMFiles,&Context->ListEntr
| y,&ReadOnlySpinLock);
70557|            }
70558|        }
70559|    }
70560| }
70561|
70562| #if 0
70563| // rob 11-11-2001 - This is a failed attempt at
| keeping
70564| // cluster server from failing over the volumes
| during a revert
70565| // operation. Cluster server still detected the
| volume offline
70566| // and failed the volume anyway.
70567|
70568|
70569| // check to see if this is one of the special
70570| // cluster files that we need to check for
70571| // when doing the revert operation. We do this so
70572| // cluster server doesnt fail the volume while we
| are
70573| // reverting
70574| PFS_FILTER_EXTENSION
| deviceExtension=(PFS_FILTER_EXTENSION)GetDeviceExtension
| (DeviceObject);
70575|
70576| if(!(deviceExtension->Virtual) &&

```

```

    | (deviceExtension->PSMStorageObject)) {
70577|     PFILTERED_EXTENSION
    | DevExt=GetFilteredExtension(deviceExtension->PSMStorageO
    | bject);
70578|
70579|     if(DevExt->IsReverting) {
70580|         DumpCreateInfo(Irp);
70581|
70582|         if(IrpSp->FileObject->FileName.Length) {
70583|
    | if(IsFileNameClusterFile(&IrpSp->FileObject->FileName)
    | ||
70584|
    | (IsFileNameRootDir(&IrpSp->FileObject->FileName))) {
70585| #ifdef DEBUG
70586|         DumpCreateInfo(Irp);
70587| #endif
70588|
70589|         WCHAR *Buffer=(WCHAR
    | *)ExAllocatePoolWithTag(PagedPool,256*sizeof(WCHAR),FILE
    | NAMETAG);
70590|         if(Buffer) {
70591|
    | RtlZeroMemory(Buffer,256*sizeof(WCHAR));
70592|
    | wcsncpy(Buffer,L"\\??\\C:\\temp");
70593|
    | wcscat(Buffer,IrpSp->FileObject->FileName.Buffer);
70594|
    | ExFreePool(IrpSp->FileObject->FileName.Buffer);
70595|
    | RtlInitUnicodeString(&IrpSp->FileObject->FileName,Buffer
    | );
70596|         Irp->IoStatus.Status =
    | STATUS_REPARSE;
70597|         Irp->IoStatus.Information =
    | IO_REPARSE;
70598|         IoCompleteRequest(Irp,
    | IO_NO_INCREMENT);
70599|         return STATUS_REPARSE;
70600|     } else {
70601|         Irp->IoStatus.Status =
    | STATUS_INSUFFICIENT_RESOURCES;
70602|         Irp->IoStatus.Information = 0;
70603|         IoCompleteRequest(Irp,
    | IO_NO_INCREMENT);
70604|         return
    | STATUS_INSUFFICIENT_RESOURCES;
70605|     }
70606|

```

```

70607|         } // is a cluster file
70608|     } // filename is valid
70609| } // is reverting
70610| } // if not virtual
70611|
70612| #endif // if 0 for cluster fix
70613|
70614| #endif
70615|     return PSManFSPassThru( DeviceObject, Irp );
70616| }
70617|
70618| #if DO_FILE_SYSTEM_FILTER
70619|
70620| STATIC
70621| NTSTATUS
70622| SfCreateCompletion(
70623|     IN PDEVICE_OBJECT DeviceObject,
70624|     IN PIRP Irp,
70625|     IN PVOID Context
70626| )
70627|
70628| /*++
70629|
70630| Routine Description:
70631|
70632|     This function is the create/open completion routine
       | for this filter
70633|     file system driver. If debugging is enabled, then
       | this function prints
70634|     the name of the file that was successfully
       | opened/created by the file
70635|     system as a result of the specified I/O request.
70636|
70637| Arguments:
70638|
70639|     DeviceObject - Pointer to the device on which the
       | file was created.
70640|
70641|     Irp - Pointer to the I/O Request Packet the
       | represents the operation.
70642|
70643|     Context - This driver's context parameter - unused;
70644|
70645| Return Value:
70646|
70647|     The function value is STATUS_SUCCESS.
70648|
70649| --*/
70650|
70651| {

```

```

70652| #define BUFFER_SIZE 1024
70653|
70654|   PIO_STACK_LOCATION IrpSp =
      | IoGetCurrentIrpStackLocation( Irp );
70655|   PFS_FILTER_EXTENSION deviceExtension =
      | (PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceObject);
70656|   NTSTATUS status;
70657|   POBJECT_NAME_INFORMATION nameInfo;
70658|   ULONG size;
70659|   PIO_STACK_LOCATION IrpSp =
      | IoGetCurrentIrpStackLocation( Irp );
70660|
70661|   | ASSERT(PsmGetObjectype(DeviceObject)==OBJECT_FS_FILTER)
      | ;
70662|   //Debug(DEBUG_SFILTER,("SFILTER: Create completion
      | routine\n"));
70663|
70664|   // if readonly volume, and the error is not found,
      | then
70665|   // it really is readonly as it create disposition
      | was OPEN_IF
70666|   if ( (deviceExtension->Virtual) &&
      | (deviceExtension->PSMStorageObject) ) {
70667|       PVDISK_EXTENSION
      | VDiskExt=(PVDISK_EXTENSION)GetDeviceExtension(deviceExte
      | nsion->PSMStorageObject);
70668|
70669|       if ( (VDiskExt->SnapShot) &&
      | pPersistentDictionary(VDiskExt->SnapShot->Dictionary)->I
      | sReadOnly() ) {
70670|           if ( FileIsReadOnly(IrpSp->FileObject) ) {
70671|               if (
      | Irp->IoStatus.Status==STATUS_OBJECT_NAME_NOT_FOUND ) {
70672|                   if (
      | IrpSp->Parameters.Create.SecurityContext->DesiredAccess
      | == FILE_OPEN ) {
70673|                       Debug(DEBUG_SFILTER,("SFILTER:
      | File didnt exist, write protected\n"));
70674|
      | IoCancelFileOpen(deviceExtension->TargetDeviceObject,Irp
      | Sp->FileObject);
70675|                       Irp->IoStatus.Status =
      | STATUS_MEDIA_WRITE_PROTECTED;
70676|                   }
70677|               }
70678|           }
70679|       }
70680|   }
70681|

```



```

70682|
70683|     if ( NT_SUCCESS(Irp->IoStatus.Status) ) {
70684|     #if 0
70685|         //
70686|         // If any debugging level is enabled, attempt
70687|         | to capture the name of the
70688|         // file that was just created/opened.
70689|         //
70690|         if ( DebugLevel & DEBUG_SFILTER ) {
70691|             if ( nameInfo =
70692|                 | (POBJECT_NAME_INFORMATION)MemAllocatePoolWithTag(
70693|                 | NonPagedPool, BUFFER_SIZE , TEMPTAG) ) {
70694|
70695|                 //
70696|                 // A buffer was successfully allocated.
70697|                 | Attempt to determine
70698|                 // whether this was a volume or a file
70699|                 | open, based on the length
70700|                 // of the file's name.  If it was a
70701|                 | volume open, then simply
70702|                 // query the name of the device.  Note
70703|                 | that it is not legal to
70704|                 // perform a relative file open using a
70705|                 | NULL name to obtain another
70706|                 // handle to the same file, so checking
70707|                 | the RelatedFileObject field
70708|                 // is unnecessary.
70709|                 //
70710|                 if ( irpSp->FileObject->FileName.Length
70711|                     | ) {
70712|                     status = ObQueryNameString(
70713|                         | irpSp->FileObject,
70714|                         | nameInfo,
70715|                         | BUFFER_SIZE,
70716|                         | &size
70717|                         | );
70718|                 } else {
70719|                     status = ObQueryNameString(
70720|                         | irpSp->FileObject->DeviceObject,
70721|                         | nameInfo,
70722|                         | BUFFER_SIZE,
70723|                         | &size
70724|                         | );
70725|                 }
70726|             }
70727|         }
70728|     }
70729| }

```

```

70718|
70719|         //
70720|         // If querying the name was successful,
70721|         | actually print the name
70722|         // on the debug terminal.
70723|         //
70724|         if ( NT_SUCCESS( status ) ) {
70725|             if (
70726|                 | irpSp->Parameters.Create.Options & FILE_OPEN_BY_FILE_ID
70727|                 | ) {
70728|                 Debug(DEBUG_SFILTER,( "SFILTER:
70729|                 | Opened %ws\\(FID)\n", nameInfo->Name.Buffer ));
70730|             } else {
70731|                 Debug(DEBUG_SFILTER,( "SFILTER:
70732|                 | Opened %ws\n", nameInfo->Name.Buffer ));
70733|             }
70734|         } else {
70735|             Debug(DEBUG_SFILTER,( "SFILTER:
70736|             | Could not get the name for %x\n", irpSp->FileObject ));
70737|         }
70738|         MemFreePool( nameInfo );
70739|     }
70740| #endif
70741| } else {
70742|     // remove the entry as close will not be called
70743|     | since the create failed
70744|     FileIsOpenOnSnapShot(IrpSp->FileObject,TRUE);
70745|     FileIsPSM(IrpSp->FileObject,TRUE);
70746| }
70747| //
70748| // Propagate the IRP pending flag.
70749| //
70750| if ( Irp->PendingReturned ) {
70751|     IoMarkIrpPending( Irp );
70752| }
70753| return STATUS_SUCCESS;
70754| }
70755|
70756| typedef struct sHelperThread {
70757|     KEVENT Event;
70758|     PDEVICE_OBJECT DeviceObject;
70759|     PIRP Irp;
70760|     ULONG State;
70761| } tHelperThread, *pHelperThread;
70762|

```

```

70761|
70762| NTSTATUS
70763| FsCtlExtendVolumeCompletionRoutine(
70764|     IN PDEVICE_OBJECT DeviceObject,
70765|     IN PIRP           Irp,
70766|     IN PVOID           Context
70767| )
70768| {
70769|     NOT_REFERENCED(DeviceObject);
70770|
70771|     pmSetEvent((PKEVENT)Context);
70772|
70773|     // keep nt from touching our irp, which will be
       | handled by person
70774|     // who wanted the event set
70775|     return STATUS_MORE_PROCESSING_REQUIRED;
70776| }
70777|
70778|
70779|
70780|
70781| STATIC
70782| NTSTATUS
70783| SfFsControl(
70784|     IN PDEVICE_OBJECT DeviceObject,
70785|     IN PIRP Irp
70786| )
70787|
70788| /*++
70789|
70790| Routine Description:
70791|
70792|     This routine is invoked whenever an I/O Request
       | Packet (IRP) w/a major
70793|     function code of IRP_MJ_FILE_SYSTEM_CONTROL is
       | encountered. For most
70794|     IRPs of this type, the packet is simply passed
       | through. However, for
70795|     some requests, special processing is required.
70796|
70797| Arguments:
70798|
70799|     DeviceObject - Pointer to the device object for
       | this driver.
70800|
70801|     Irp - Pointer to the request packet representing
       | the I/O request.
70802|
70803| Return Value:
70804|

```

```

70805|   The function value is the status of the operation.
70806|
70807| --*/
70808|
70809| {
70810|   NTSTATUS status;
70811|   PIO_STACK_LOCATION irpSp =
       | IoGetCurrentIrpStackLocation( Irp );
70812|   PDEVICE_OBJECT deviceObject;
70813|   PFS_FILTER_EXTENSION deviceExtension =
       | (PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceObject);
70814|
70815|   PAGED_CODE();
70816|
70817| #ifdef DEBUG
70818|   #if DO_ALL_SFILTER
70819|
       | if(irpSp->Parameters.DeviceIoControl.IoControlCode!=FSCT
       | L_IS_VOLUME_MOUNTED) {
70820|       Debug(DEBUG_SFILTER,("SFILTER: FS Control,
       | mj=%08x, mn=%08x, fs=%08x, fl=%08x, '%s'\n",
70821|       irpSp->MajorFunction,
70822|       irpSp->MinorFunction,
70823|
       | irpSp->Parameters.DeviceIoControl.IoControlCode,
70824|       irpSp->Flags,
70825|       File_GetFSCTLFunctionName(
70826|       irpSp->MinorFunction,
70827|
       | irpSp->Parameters.DeviceIoControl.IoControlCode
70828|       ));
70829|   }
70830|   #endif /*DO_ALL_SFILTER*/
70831| #endif /*DEBUG*/
70832|
70833|   //
70834|   // Begin by determining the minor function code for
       | this file system control
70835|   // function.
70836|   //
70837|
70838| // && (!(irpSp->Flags & SL_ALLOW_RAW_MOUNT))
70839|   if ( (irpSp->MinorFunction == IRP_MN_MOUNT_VOLUME
       | )) {
70840|
70841|       #if DO_ALL_SFILTER
70842|       Debug(DEBUG_SFILTER,("SFILTER: Mount Volume
       | Request\n"));
70843|       #endif /*DO_ALL_SFILTER*/
70844|       //

```

```

70845|      // This is a mount request. Create a device
| object that can be
70846|      // attached to the file system's volume device
| object if this request
70847|      // is successful. Note that it is possible
| that there is a device
70848|      // object already on the FsDeviceQueue as the
| result of a mini-file
70849|      // system recognizer having recognized a
| volume. If so, then attempt
70850|      // to use it instead.
70851|      //
70852|
70853|      deviceObject = (PDEVICE_OBJECT) NULL;
70854| #if 0
70855|      /*
70856|      10-18-2001 Rob - For some reason i havent
| figured out, this code is
70857|      providing the wrong object to Mount, which
| is causing us to attach to
70858|      the wrong stack. Basically you can see
| this problem if you install
70859|      another file system filter driver over us.
| You will get a BSOD of
70860|      NO_MORE_IRP_STACK_LOCATIONS. I tracked
| this down by using srecog.sys
70861|      which is a very simple file system
| recognizer. Since i took this code
70862|      from the windows 2000 sfilter, i am not
| sure why it is wrong. Also
70863|      according to the news groups, the windows
| 2000 sfilter i took this code
70864|      from is confusing (if not wrong), so the
| windows XP sfilter does it a
70865|      different way. Maybe one day, we need to
| see how.
70866|
70867|
70868|      This problem occurs when a file system
| recognizer issues a load command.
70869|      By not reusing this object, it will be
| leaked. Since this occurs only when
70870|      the file system is being loaded by a
| recognizer, it should be ok. Also if
70871|      recognizer goes away, so will the object.
70872|      */
70873|      if ( !IsListEmpty( &FsDeviceQueue ) ) {
70874|
70875|          PLIST_ENTRY entry;
70876|

```

```

70877|         FsRtlEnterFileSystem();
70878|         ExAcquireResourceExclusive( &FsLock, TRUE
70879|         | );
70879|         if ( !IsListEmpty( &FsDeviceQueue ) ) {
70880|
70881|             //
70882|             // There is a device object on the
70883|             | device queue that may be
70883|             // reusable. Remove it from the queue
70884|             | and check its reference
70884|             // count; if it is zero, then it can
70885|             | be used, otherwise, put
70885|             // it back. Note that device objects
70886|             | are inserted onto the tail
70886|             // of the queue, so those at the head
70887|             | should be usable since the
70887|             // reference count is decremented as
70888|             | soon as the top level
70888|             // driver's completion routine returns.
70889|             //
70890|
70891|             entry = FsDeviceQueue.Flink;
70892|             deviceObject = CONTAINING_RECORD(
70893|                 entry,
70894|                 | DEVICE_OBJECT,
70895|                 | Queue.ListEntry
70896|                 );
70897|             if ( !deviceObject->ReferenceCount ) {
70898|                 RemoveHeadList( &FsDeviceQueue );
70899|                 | ASSERT(PsmGetObjectTypeInfo(deviceObject)==OBJECT_FS_FILTER)
70900|                 | ;
70900|                 status = STATUS_SUCCESS;
70901|             } else {
70902|                 deviceObject = (PDEVICE_OBJECT)
70903|                 | NULL;
70903|             }
70904|         }
70905|         ExReleaseResource(&FsLock );
70906|         FsRtlExitFileSystem();
70907|     }
70908| #endif
70909|
70910|     if ( !deviceObject ) {
70911|
70912| #ifdef DEBUG_EXTENSION
70913|         ULONG SizeOfDeviceExt =
70914|         | sizeof(DEVICE_EXTENSION);

```

```

70914| #else
70915|         ULONG SizeOfDeviceExt = sizeof(
70916|             | FS_FILTER_EXTENSION);
70917| #endif
70917|         status = IoCreateDevice(
70918|             PSMAN_DRIVER_OBJECT,
70919|             SizeOfDeviceExt,
70920|             (PUNICODE_STRING)
70921|             | NULL,
70922|             | FILE_DEVICE_DISK_FILE_SYSTEM,
70923|             0,
70924|             FALSE,
70925|             &deviceObject
70926|             );
70927|         if ( NT_SUCCESS(status) ) {
70928|             PFS_FILTER_EXTENSION Ext;
70929|             #if DEBUG_ALL_SFILTER
70930|                 Debug(DEBUG_SFILTER,("SFILTER:
70931|                 | Created new FS filter %08x\n",deviceObject));
70932|             #endif /*DEBUG_ALL_SFILTER*/
70933|
70934|             | ((PDEVICE_EXTENSION)(deviceObject->DeviceExtension))->Ob
70935|             | jectType = OBJECT_FS_FILTER;
70936|
70937|             | ((PDEVICE_EXTENSION)(deviceObject->DeviceExtension))->Re
70938|             | alDeviceExtension =
70939|             | MemAllocatePoolWithTag(NonPagedPool,FS_FILTER_EXTENSION_
70940|             | SIZE,DEVEXTTAG);
70941|
70942|             | RtlZeroMemory(((PDEVICE_EXTENSION)(deviceObject->DeviceE
70943|             | xtension))->RealDeviceExtension,FS_FILTER_EXTENSION_SIZE
70944|             | );
70945| #endif
70946|
70947|             Ext =
70948|             | (PFS_FILTER_EXTENSION)GetDeviceExtension(deviceObject);
70949|             Ext->DeviceObject = deviceObject;
70950|             Ext->DriverObject = PSMAN_DRIVER_OBJECT;
70951|             Ext->ObjectType = OBJECT_FS_FILTER;
70952|             Ext->Attached = FALSE;
70953|             Ext->TargetDeviceObject = NULL;
70954|             Ext->PSMStorageObject = NULL;
70955|             Ext->Virtual = FALSE;
70956|             Ext->FileSystem = FALSE;
70957|         }
70958|     } else {

```

```

70950|         Debug(DEBUG_SFILTER,("SFILTER: Reusing FS
| filter %08x\n",deviceObject));
70951|     }
70952|
70953|     if ( NT_SUCCESS( status ) && deviceObject ) {
70954|
70955|         IoCopyCurrentIrpStackLocationToNext( Irp );
70956|
70957|         //
70958|         // Set the address of the completion
| routine for this mount request
70959|         // to be the mount completion routine and
| pass along the address
70960|         // of the specified device object as its
| context.
70961|         //
70962|         // Also, pass a pointer to the real device
| object from the VPB so
70963|         // that a remount VPB can be located if
| necessary (see comments in
70964|         // the mount completion routine).
70965|         //
70966|
70967|         IoSetCompletionRoutine(
70968|             Irp,
70969|             SfMountCompletion,
70970|             deviceObject,
70971|             TRUE,
70972|             TRUE,
70973|             TRUE
70974|         );
70975|
70976| #if DO_ALL_SFILTER
70977|         Debug(DEBUG_SFILTER,("SFILTER: 1 Sending
| mount to lower driver\n"));
70978|         Debug(DEBUG_SFILTER,("SFILTER: 1 MyDo=%08x,
| Mydo=%08x, mvpb=%08x, mdo=%08x, vrd=%08x,
| vdo=%08x,vfl=%08x\n",
70979|             DeviceObject,
70980|             deviceObject,
70981|
| irpSp->Parameters.MountVolume.Vpb,
70982|
| irpSp->Parameters.MountVolume.DeviceObject,
70983|
| irpSp->Parameters.MountVolume.Vpb->RealDevice,
70984|
| irpSp->Parameters.MountVolume.Vpb->DeviceObject,
70985|
| irpSp->Parameters.MountVolume.Vpb->Flags

```



```

70986|         ));
70987| #endif /*DO_ALL_SFILTER*/
70988|
70989|         irpSp->Parameters.MountVolume.DeviceObject
70990|         | = irpSp->Parameters.MountVolume.Vpb->RealDevice;
70991|         PDEVICE_OBJECT PSMStorageObject =
70992|         | GetPSMStorageFilterObject(irpSp->Parameters.MountVolume.
70993|         | Vpb);
70994|         if ( PSMStorageObject ) {
70995|             if ( PsmGetObjectTypes(PSMStorageObject)
70996|             | == OBJECT_FILTEREDDISK ) {
70997|                 PFILTERED_EXTENSION
70998|                 | DevExt=(PFILTERED_EXTENSION)GetDeviceExtension(PSMStorage
70999|                 | eObject);
71000|
71001|                 #if DO_ALL_SFILTER
71002|                 Debug(DEBUG_SFILTER,("SFILTER:
71003|                 | 1 Lower object is filtered disk %08x,
71004|                 | mounted=%d\n",PSMStorageObject,DevExt->IsMounted));
71005|                 #endif /*DO_ALL_SFILTER*/
71006|                 // we will reload the snapshots
71007|                 | incase it changed from the last time
71008|                 if(!DevExt->IsPhysical) &&
71009|                 | (!DevExt->IsReverting)) {
71010|                     if(!GlobalData->ShutDownCalled)
71011|                     | {
71012|
71013|                     // load value from registry
71014|                     | as it may not have been loaded yet
71015|                     if (DevExt->FileSystem ==
71016|                     | FILE_SYSTEM_UNKNOWN) {
71017|                         // see about loading
71018|                         WCHAR *VolumeReg =
71019|                         | GetPerVolumeRegistry(PSMStorageObject);
71020|                         if(VolumeReg) {
71021|                             UNICODE_STRING Str;
71022|                             | RtlInitUnicodeString( &Str, VolumeReg);
71023|                             | Reg_GetULONGKey(&Str,L"FileSystem",FILE_SYSTEM_UNKNOWN,&
71024|                             | DevExt->FileSystem);
71025|                             | FreePerVolumeRegistry(VolumeReg);
71026|                         }
71027|                     }
71028|
71029|                     // try and prevent loading
71030|                     | of snapshots multiple times

```

```

71017|                // ie we get a mount
| request for a NTFS volume multiple times:
71018|                //    FastFat
71019|                //    DFS
71020|                //    NTFS
71021|                // so this code should keep
| us from loading the index 2 or more times.
71022|                if( ((DevExt->FileSystem ==
| FILE_SYSTEM_NTFS) && (deviceExtension->IsNtfs)) ||
71023|                ((DevExt->FileSystem ==
| FILE_SYSTEM_FAT) && (deviceExtension->IsFat)) ||
71024|                ((DevExt->FileSystem ==
| FILE_SYSTEM_UNKNOWN))
71025|                ) {
71026|
71027|                #if DO_ALL_SFILTER
71028|                | Debug(DEBUG_SFILTER,("SFILTER: 1 Lower object not
| physical, attempting to load snapshots\n"));
71029|                #endif
| /*DO_ALL_SFILTER*/
71030|                | if(PersistentDictionary::CheckIfLoadNeeded(PSMStorageObj
| ect)) {
71031|
71032|                #ifdef
| DO_TIMING_TEST
71033|                /*lint -save
| -e740 */
71034|                | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_LOAD
| ING_SNAPSHOTS,0,NULL,0,NULL,0);
71035|                /*lint -restore
| */
71036|                #endif
71037|
71038|                NTSTATUS Status =
| PersistentDictionary::LoadSnapShotsForVolume
| (PSMStorageObject,TRUE,NULL );
71039|
71040|                #ifdef
| DO_TIMING_TEST
71041|                /*lint -save
| -e740 */
71042|                | LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,PSM_DONE
| _LOADING_SNAPSHOTS,0,NULL,0,NULL,0);
71043|                /*lint -restore
| */
71044|                #endif

```

```

71045|
71046|             #if DO_ALL_SFILTER
71047|         | Debug(DEBUG_SFILTER,("SFILTER: 1 load snapshots
71048|         | returned %08x\n",Status));
71049|             #endif
71050|         | /*DO_ALL_SFILTER*/
71051|         | // if out of
71052|         | memory, dont allow the volume to come online
71053|         | if((Status==STATUS_INSUFFICIENT_RESOURCES) ||
71054|         | (Status==PSM_ERROR_OUT_OF_MEMORY)) {
71055|         | Debug(DEBUG_SFILTER,("SFILTER: Error! out of memory,
71056|         | failing mount!\n"));
71057|         | Irp->IoStatus.Status = STATUS_INSUFFICIENT_RESOURCES;
71058|         | Irp->IoStatus.Information = 0;
71059|         | IoCompleteRequest( Irp, IO_NO_INCREMENT );
71060|         | return
71061|         | STATUS_INSUFFICIENT_RESOURCES;
71062|         | }
71063|         | else {
71064|         | // see if a revert
71065|         | needs to occur as we didnt call LoadSnapShotsForVolume
71066|         | if(DevExt->Cache.Header) {
71067|         |     if (
71068|         |         DevExt->Cache.Header->RevertInfo.SnapShotSequenceNumber
71069|         |         != 0 ) {
71070|         |         ULONG
71071|         |         Opened=0;
71072|         |         if(!DevExt->OpenCloseAcquired) {
71073|         |             if (
71074|         |                 AcquireOpenCloseResourceOnly(NULL)!=STATUS_WAIT_0 ) {
71075|         |                 goto Leave;
71076|         |             }
71077|         |             Opened
71078|         |             = TRUE;
71079|         |         DevExt->OpenCloseAcquired = TRUE;

```

```

71073|             }
71074|
71075|     | RevertCheckAtVolumeMount(PSMStorageObject);
71076|
71077|             if(Opened)
71078|     | {
71079|     | DevExt->OpenCloseAcquired = FALSE;
71080|     | ReleaseOpenCloseResource();
71081| Leave:
71082|             NOTHING;
71083|     }
71084|     }
71085|
71086|     }
71087|     } else {
71088|         // wrong file system
71089|     | for us
71090|     | Debug(DEBUG_SFILTER,("SFILTER: mount request to wrong
71091|     | volume type, fs=%d, ntfs=%d, fat=%d,
71092|     | raw=%d\n",DevExt->FileSystem,deviceExtension->IsNtfs,dev
71093|     | iceExtension->IsFat,deviceExtension->IsRaw));
71094|
71095|     // Since we know this
71096|     | is wrong, tell the system
71097|
71098|     Irp->IoStatus.Status =
71099|     | STATUS_UNRECOGNIZED_VOLUME;
71100|
71101|     | Irp->IoStatus.Information = 0;
71102|
71103|     IoCompleteRequest( Irp,
71104|     | IO_NO_INCREMENT );
71105|
71106|     return
71107|     | STATUS_UNRECOGNIZED_VOLUME;
71108|     }
71109|     } else {
71110|         Debug(DEBUG_DICT,("SFILTER:
71111|         | shutdown called, not loading snapshots\n"));
71112|     }
71113|     } else {
71114|         if(DevExt->IsReverting) {
71115|
71116|         | Debug(DEBUG_SFILTER,("SFILTER: mount request while
71117|         | reverting\n"));

```

```

71106|
71107|             // If reverting this
| volume, do NOT allow mount requests to complete
71108|
71109|             Irp->IoStatus.Status =
| STATUS_ACCESS_DENIED;
71110|             Irp->IoStatus.Information =
| 0;
71111|
71112|             IoCompleteRequest( Irp,
| IO_NO_INCREMENT );
71113|
71114|             return
| STATUS_ACCESS_DENIED;
71115|         } else {
71116|             #if DO_ALL_SFILTER
71117|             | Debug(DEBUG_SFILTER,("SFILTER: 1 Lower object
| physical\n"));
71118|             #endif /*DO_ALL_SFILTER*/
71119|         }
71120|     }
71121| } else if (
| PsmGetObjectTypes(PsmStorageObject) ==
| OBJECT_VIRTUALDISK ) {
71122|     #if DO_ALL_SFILTER
71123|     | Debug(DEBUG_SFILTER,("SFILTER:
| 1 Lower object is virtual disk
| %08x\n",PsmStorageObject));
71124|     #endif /*DO_ALL_SFILTER*/
71125|     PVDISK_EXTENSION VDiskExt =
| GetVDiskExtension(PsmStorageObject);
71126|     if ( VDiskExt->MountDisabled ) {
71127|         // We can get here if, for
| example, we are currently undoing virtual writes
71128|         | Debug(DEBUG_SFILTER,("SFILTER:
| mount request to virtual volume while mount
| disabled\n"));
71129|         Irp->IoStatus.Status =
| STATUS_ACCESS_DENIED;
71130|         Irp->IoStatus.Information = 0;
71131|         IoCompleteRequest( Irp,
| IO_NO_INCREMENT );
71132|         return STATUS_ACCESS_DENIED;
71133|     }
71134|
71135|     if(VDiskExt->PSMDevice) {
71136|         PFILTERED_EXTENSION
| DevExt=(PFILTERED_EXTENSION)GetDeviceExtension(VDiskExt-
| >PSMDevice);

```

```

71137|             ASSERT(DevExt->ObjectType ==
| OBJECT_FILTEREDDISK);
71138|
71139|             if(DevExt->IsReverting) {
71140|                 // If reverting this
| volume, do NOT allow mount requests to complete
71141|
| Debug(DEBUG_SFILTER,("SFILTER: mount request to virtual
| volume while reverting\n"));
71142|                 Irp->IoStatus.Status =
| STATUS_ACCESS_DENIED;
71143|                 Irp->IoStatus.Information =
| 0;
71144|                 IoCompleteRequest( Irp,
| IO_NO_INCREMENT );
71145|                 return
| STATUS_ACCESS_DENIED;
71146|             }
71147|         }
71148|     } else {
71149|         #ifdef DEBUG
71150|             Debug(DEBUG_SFILTER,("SFILTER: 1
| Error! Unknown lower object!!!!!!!!!!!!!!!!\n"));
71151|             DbgBreakPoint();
71152|         #endif
71153|     }
71154| } else {
71155|     // this can happen when the volume
| being mounted is not a volume we filter
71156|     // for example, A: or B:
71157| }
71158| } else {
71159|     Debug(DEBUG_SFILTER,("SFILTER: 1 Unable to
| create device, not attaching to volume!!!!!!!!\n"));
71160|
71161|     //
71162|     // Something went wrong so this volume
| cannot be filtered. Simply
71163|     // allow the system to continue working
| normally, if possible.
71164|     //
71165|     IoSkipCurrentIrpStackLocation( Irp );
71166| }
71167| } else
71168|     if ( irpSp->MinorFunction ==
| IRP_MN_LOAD_FILE_SYSTEM ) {
71169|         #if DO_ALL_SFILTER
71170|             Debug(DEBUG_SFILTER,("SFILTER: Load File
| System\n"));
71171|         #endif /*DO_ALL_SFILTER*/

```

```

71172|
71173|    //
71174|    // This is a load file system request being
    | sent to a mini-file system
71175|    // recognizer driver. Detach from the file
    | system now, and set
71176|    // the address of a completion routine in case
    | the function fails, in
71177|    // which case a reattachment needs to occur.
    | Likewise, if the function
71178|    // is successful, then the device object needs
    | to be deleted.
71179|    //
71180|
71181|    IoCopyCurrentIrpStackLocationToNext( Irp );
71182|
71183|    IoDetachDevice(
    | deviceExtension->TargetDeviceObject );
71184|    deviceExtension->Attached = FALSE;
71185|
71186|    IoSetCompletionRoutine(
71187|        Irp,
71188|        SfLoadFsCompletion,
71189|        DeviceObject,
71190|        TRUE,
71191|        TRUE,
71192|        TRUE
71193|    );
71194| } else
71195|     if (
    | PsmGetObjectype(DeviceObject)!=OBJECT_FS_FILTER ) {
71196|
71197|         #if DO_ALL_SFILTER
71198|             Debug(DEBUG_SFILTER,("SFILTER: Request
    | directed to object and not filter\n"));
71199|         #endif /*DO_ALL_SFILTER*/
71200|         //
71201|         // If this device object is a primary rather
    | than a pass-through,
71202|         // then this is an invalid request.
71203|         //
71204|
71205|         Irp->IoStatus.Status =
    | STATUS_INVALID_DEVICE_REQUEST;
71206|         Irp->IoStatus.Information = 0;
71207|
71208|         IoCompleteRequest( Irp, IO_NO_INCREMENT );
71209|
71210|         return STATUS_INVALID_DEVICE_REQUEST;
71211|     } else

```

```

71212|     if ( irpSp->MinorFunction ==
| IRP_MN_USER_FS_REQUEST ) {
71213|     NTSTATUS Status;
71214|     switch (
| irpSp->Parameters.DeviceIoControl.IoControlCode ) {
71215|     case FSCTL_DISMOUNT_VOLUME: {
71216|         if ( !deviceExtension->Virtual ) {
71217|             PFILTERED_EXTENSION d =
| (PFILTERED_EXTENSION)GetDeviceExtension(deviceExtension-
| >PSMStorageObject);
71218|             BOOLEAN
| DirectState=d->DoDirectIO;
71219|
71220|             Debug(DEBUG_SFILTER,("SFILTER:
| Dismount Volume Request for real volume, do=%08x,
| irp=%08x,
| psm=%08x\n",DeviceObject,Irp,deviceExtension->PSMStorage
| Object));
71221|
71222|             PersistentDictionary::MiniUnloadSnapShotsForVolume(devic
| eExtension->PSMStorageObject);
71223|             ULONG PrevState=d->IsMounted;
71224|
71225|             d->IsMounted = FALSE;
71226|             Debug(DEBUG_SFILTER,("SFILTER:
| Just set IsMounted to FALSE for DevExt=%08x,
| DevObj=%08x
| (PrevState=%08x)\n",d,deviceExtension->PSMStorageObject,
| PrevState));
71227|
71228|             // handle it synchronous
71229|
71230|             // we need to load snapshots
| back up if the dismount fails for whatever reason
71231|             Status =
| PSMANForwardIrpSynchronous(DeviceObject,Irp);
71232|
71233|             Debug(DEBUG_SFILTER,("SFILTER:
| PSMANForwardIrpSynchronous returned %08x irp=%08x
| during dismount,
| fo=%08x\n",Status,Irp->IoStatus.Status,Irp->Tail.Overlay
| .OriginalFileObject));
71234|
71235|             // for some reason, we can't
| fathom, when a dismount fails,
71236|             // the status is still
| STATUS_SUCCESS. In this case, we will
71237|             // high jack the current irp's
| file object, and ask the file system

```



```

71238|                // if it thinks it is
| dismantled or not.
71239|                // it returns STATUS_SUCCESS if
| mounted
71240|                // or STATUS_VOLUME_DISMOUNTED
| if dismantled
71241|                // another thing we could do is
| intercept FSCTL_IS_VOLUME_MOUNTED
71242|                // and update our status.
71243|                ULONG status2;
71244|                BOOLEAN PutBack = TRUE;
71245|
71246|                if(Status==STATUS_SUCCESS) {
71247|                    PFILE_OBJECT FO =
| irpSp->FileObject;
71248|                    if(!FO) {
71249|                        FO =
| Irp->Tail.Overlay.OriginalFileObject;
71250|                    }
71251|                    ASSERT(FO);
71252|
| Debug(DEBUG_SFILTER,("SFILTER: Objects = %08x, %08x,
| using
| %08x\n",Irp->Tail.Overlay.OriginalFileObject,irpSp->File
| Object,FO));
71253|                status2 =
| FS_IsVolumeMounted(FO);
71254|
| Debug(DEBUG_SFILTER,("SFILTER: IsVolumeMounted =
| %08x\n",status2));
71255|                PutBack = (status2 ==
| STATUS_SUCCESS); //PutBack is TRUE only if volume is
| mounted
71256|                } else {
71257|                    status2 = Status;
71258|                }
71259|
71260|                if(PutBack) {
71261|                    // The dismount failed if
| we get here.
71262|                    d->IsMounted = PrevState;
71263|                    Debug(DEBUG_SFILTER,("Just
| set IsMounted back to %08x for DevExt=%08x,
| DevObj=%08x\n",PrevState,d,deviceExtension->PSMStorageOb
| ject));
71264|
71265|                    // if previous state was
| mounted then remap in virtual drives
71266|                    if((PrevState==TRUE) &&
| (PersistentDictionary::GetSystemReady()) &&

```

```

    | (!d->IsReverting)) {
71267|             HANDLE TempHandle;
71268|
71269|
    | pmStartThread(PersistentDictionary::Part2OfRebuildForVol
    | ume,deviceExtension->PSMStorageObject,&TempHandle);
71270|
    | ZwWaitForSingleObject(TempHandle,FALSE,NULL);
71271|             ZwClose(TempHandle);
71272|         } else {
71273|             // got set to true in
    | MiniUnload
71274|             d->DoDirectIO =
    | DirectState;
71275|         }
71276|     }
71277|
71278|     IoCompleteRequest(Irp,
    | IO_NO_INCREMENT);
71279|     return Status;
71280| } else {
71281|     Debug(DEBUG_SFILTER,("SFILTER:
    | Dismount Volume Request for virtual volume\n"));
71282|     IoSkipCurrentIrpStackLocation(
    | Irp );
71283| }
71284|     break;
71285| }
71286|
71287|     case FSCTL_EXTEND_VOLUME : {
71288|
71289|         // extends a dynamic volume. We need to
    | intercept a successful return in order to extend the
    | volume bit map.
71290|         Debug(DEBUG_SFILTER,("SFILTER: Extend
    | Volume Request\n"));
71291|
71292|         if ( (!deviceExtension->Virtual) &&
    | (deviceExtension->PSMStorageObject) ) {
71293|             PFILTERED_EXTENSION Ext =
    | (PFILTERED_EXTENSION)GetDeviceExtension(deviceExtension-
    | >PSMStorageObject);
71294|
71295|             //
71296|             int Switch=1;    // 0 = Fail
71297|             // 1 = Catch
    | Return
71298|             // 2 = passthru
71299|
71300|             if((Ext->PSMed) &&

```

```

    | (PersistentDictionary::DoFreeSpaceChecks())) {
71301|         switch (Switch) {
71302|             case 0 :
71303|
71304|         | Debug(DEBUG_SFILTER,("SFILTER: returning error to
71305|         | caller\n" ));
71306|         | Irp->IoStatus.Status =
71307|         | STATUS_INVALID_DEVICE_REQUEST;
71308|         | Irp->IoStatus.Information = 0;
71309|         | IoCompleteRequest( Irp,
71310|         | IO_NO_INCREMENT );
71311|         | return
71312|         | STATUS_INVALID_DEVICE_REQUEST;
71313|         case 1 : {
71314|             KEVENT Event;
71315|             PLARGE_INTEGER
71316|             | NewSize=(PLARGE_INTEGER)Irp->AssociatedIrp.SystemBuffer;
71317|             // Extend the bitmap
71318|             Status =
71319|             | ExtendFreeSpaceBitmaps(
71320|             | deviceExtension->PSMStorageObject, *NewSize );
71321|             if(NT_SUCCESS(Status))
71322|             | {
71323|             | Debug(DEBUG_SFILTER,("Success extending bitmaps\n"));
71324|             | // modify what we
71325|             | know about the size of this partition
71326|             | Ext->Pi.PartitionLength.QuadPart = NewSize->QuadPart *
71327|             | 512;
71328|             } else {
71329|             | Debug(DEBUG_SFILTER,("Error %08x extending
71330|             | bitmaps\n",Status));
71331|             }
71332|             // Send the command to
71333|             | the file system
71334|             | KeInitializeEvent(&Event,NotificationEvent,FALSE);
71335|             | Debug(DEBUG_DEVSUP,("Extend: Sending extend to lower
71336|             | drivers.\n"));
71337|
71338|
71339|

```

```

    | IoCopyCurrentIrpStackLocationToNext( Irp );
71330|
71331|         IoSetCompletionRoutine(
71332|             Irp,
71333|
    | FsCtlExtendVolumeCompletionRoutine,
71334|             &Event,
71335|             TRUE,
71336|             TRUE,
71337|             TRUE
71338|         );
71339|
71340|         Status = IoCallDriver(
    | deviceExtension->TargetDeviceObject, Irp );
71341|
71342|         if (Status ==
    | STATUS_PENDING) {
71343|
    | ASSERT(KernelGetCurrentIrp() < DISPATCH_LEVEL);
71344|
    | pmWaitForSingleObject(&Event,NULL);
71345|
71346|         Status =
    | Irp->IoStatus.Status;
71347|
    | Debug(DEBUG_DEVSUP,("Extend: Extend finished with wait.
    | IoStatus=%08x,%08x\n",Irp->IoStatus.Status,Irp->IoStatus
    | .Information));
71348|         } else {
71349|
    | Debug(DEBUG_DEVSUP,("Extend: Extend finished without
    | wait. Status=%08x,
    | IoStatus=%08x,%08x\n",Status,Irp->IoStatus.Status,Irp->I
    | oStatus.Information));
71350|         }
71351|         // finish the request
71352|         if (
    | Irp->PendingReturned ) {
71353|             IoMarkIrpPending(
    | Irp );
71354|         }
71355|
71356|
    | IoCompleteRequest(Irp,IO_NO_INCREMENT);
71357|         return Status;
71358|     }
71359|     case 2 :
71360|
    | IoSkipCurrentIrpStackLocation( Irp );
71361|         break;

```

```

71362|             default:
71363|
71364|             | Debug(DEBUG_SFILTER,("SFILTER: unknown switch\n" ));
71365|             | IoSkipCurrentIrpStackLocation( Irp );
71366|             break;
71367|             } // switch
71368|             } // psmmed
71369|             } // not virtual
71370|             } // case
71371|         case FSCTL_GET_NTFS_VOLUME_DATA :
71372|             IoSkipCurrentIrpStackLocation( Irp );
71373|             break;
71374|         case FSCTL_GET_NTFS_FILE_RECORD:
71375|             IoSkipCurrentIrpStackLocation( Irp );
71376|             break;
71377|         case FSCTL_GET_VOLUME_BITMAP: {
71378| #if 0
71379|             if ( deviceExtension->Virtual ) {
71380|                 InvalidDeviceRequest:
71381|                 // dont allow this on virtual
71382|                 | volumes
71383|                 Irp->IoStatus.Status =
71384|                 | STATUS_INVALID_DEVICE_REQUEST;
71385|                 Irp->IoStatus.Information = 0;
71386|                 IoCompleteRequest(Irp,
71387|                 | IO_NO_INCREMENT);
71388|                 return
71389|                 | STATUS_INVALID_DEVICE_REQUEST;
71390|             }
71391|             pHelperThread
71392|             | Context=(pHelperThread)MemAllocatePoolWithTag(NonPagedPo
71393|             | ol,sizeof(tHelperThread),TEMPTAG);
71394|             if ( !Context ) {
71395|                 Irp->IoStatus.Status =
71396|                 | STATUS_INSUFFICIENT_RESOURCES;
71397|                 Irp->IoStatus.Information = 0;
71398|                 IoCompleteRequest(Irp,
71399|                 | IO_NO_INCREMENT);
71400|                 return
71401|                 | STATUS_INSUFFICIENT_RESOURCES;
71402|             }
71403|
71404|             | KeInitializeEvent(&Context->Event,NotificationEvent,FALS
71405|             | E);
71406|             Context->Irp = Irp;
71407|             Context->DeviceObject =

```

```

    | DeviceObject;
71399|
71400|         HANDLE TempHandle;
71401|         Status =
    | pmStartThread((PKSTART_ROUTINE)GetVolumeBitMapThread,Con
    | text,&TempHandle);
71402|         if ( !NT_SUCCESS(Status) ) {
71403|             MemFreePool(Context);
71404|             Irp->IoStatus.Status = Status;
71405|             Irp->IoStatus.Information = 0;
71406|             IoCompleteRequest(Irp,
    | IO_NO_INCREMENT);
71407|             return Status;
71408|         }
71409|         ZwClose(TempHandle);
71410|
71411|
71412|         // lets intercept this request
71413|
    | IoCopyCurrentIrpStackLocationToNext( Irp );
71414|
71415|         IoSetCompletionRoutine(
71416|             Irp,
71417|
    | SfHelperThreadCompletion,
71418|             Context,
71419|             TRUE, //
    | invoke on success
71420|             TRUE, //
    | invoke on error
71421|             TRUE //
    | invoke on cancel
71422|             );
71423|         break;
71424| #else
71425|         IoSkipCurrentIrpStackLocation( Irp );
71426|         break;
71427| #endif
71428|     }
71429|     case FSCTL_GET_RETRIEVAL_POINTERS:
71430|         IoSkipCurrentIrpStackLocation( Irp );
71431|         break;
71432|     case FSCTL_MOVE_FILE:
71433|         if ( deviceExtension->Virtual ) {
71434|             // dont allow this on virtual
    | volumes
71435|             Irp->IoStatus.Status =
    | STATUS_INVALID_DEVICE_REQUEST;
71436|             Irp->IoStatus.Information = 0;
71437|             IoCompleteRequest(Irp,

```

```

    | IO_NO_INCREMENT);
71438|         return
    | STATUS_INVALID_DEVICE_REQUEST;
71439|     } else {
71440|         // dont allow our file to be
    | defragged for now.
71441|         // later on, we may allow it, and
    | modify our maps directly
71442|
    | if(FileIsPSM(irpSp->FileObject,FALSE)) {
71443|         Irp->IoStatus.Status =
    | STATUS_ACCESS_DENIED;
71444|         Irp->IoStatus.Information = 0;
71445|         IoCompleteRequest(Irp,
    | IO_NO_INCREMENT);
71446|         return STATUS_ACCESS_DENIED;
71447|     } else {
71448|         // until we allow defrag and
    | PSM to work together, lets
71449|         // deny all defrags while there
    | are snapshots on this volume
71450|         PFILTERED_EXTENSION d =
    | (PFILTERED_EXTENSION)GetDeviceExtension(deviceExtension-
    | >PSMStorageObject);
71451|         if(d->PSMed) {
71452|             Irp->IoStatus.Status =
    | STATUS_ACCESS_DENIED;
71453|             Irp->IoStatus.Information =
    | 0;
71454|             IoCompleteRequest(Irp,
    | IO_NO_INCREMENT);
71455|             return
    | STATUS_ACCESS_DENIED;
71456|         }
71457|     }
71458| }
71459| IoSkipCurrentIrpStackLocation( Irp );
71460| break;
71461| default:
71462|     // send it down
71463|     IoSkipCurrentIrpStackLocation( Irp );
71464| }
71465| /*
71466| } else if ( irpSp->MinorFunction ==
    | IRP_MN_QUERY_DIRECTORY ) {
71467|     // We intercept directory requests solely for
    | removing nested snapshot
71468|     // directories on a virtual volume. All other
    | requests we allow to pass through
71469|     // untouched.

```

```

71470| */
71471| } else {
71472|
71473|     //Debug(DEBUG_SFILTER,("SFILTER: Sending
    | request down\n"));
71474|     //
71475|     // Simply pass this file system control request
    | through.
71476|     //
71477|
71478|     IoSkipCurrentIrpStackLocation( Irp );
71479| }
71480|
71481| //
71482| // Any special processing has been completed, so
    | simply pass the request
71483| // along to the next driver.
71484| //
71485|
71486| return IoCallDriver(
    | deviceExtension->TargetDeviceObject, Irp );
71487| }
71488|
71489| STATIC
71490| VOID
71491| SfFsNotification(
71492|     IN PDEVICE_OBJECT DeviceObject,
71493|     IN BOOLEAN FsActive
71494| )
71495|
71496| /*++
71497|
71498| Routine Description:
71499|
71500| This routine is invoked whenever a file system has
    | either registered or
71501|   unregistered itself as an active file system.
71502|
71503| For the former case, this routine creates a device
    | object and attaches it
71504|   to the specified file system's device object. This
    | allows this driver
71505|   to filter all requests to that file system.
71506|
71507| For the latter case, this file system's device
    | object is located,
71508|   detached, and deleted. This removes this file
    | system as a filter for
71509|   the specified file system.
71510|

```



```

71511| Arguments:
71512|
71513| DeviceObject - Pointer to the file system's device
    | object.
71514|
71515| FsActive - Ffolean indicating whether the file
    | system has registered
71516| (TRUE) or unregistered (FALSE) itself as an
    | active file system.
71517|
71518| Return Value:
71519|
71520| None.
71521|
71522| --*/
71523|
71524| {
71525|     NTSTATUS status;
71526|     PDEVICE_OBJECT deviceObject;
71527|     PDEVICE_OBJECT nextAttachedDevice;
71528|     PDEVICE_OBJECT fsDevice;
71529|
71530|     PAGED_CODE();
71531|
71532|     Debug(DEBUG_SFILTER,("SFILTER: Notification
    | routine, Device=%08x, active=%d,
    | fs='%wZ'\n",DeviceObject,FsActive,&DeviceObject->DriverO
    | bject->DriverName));
71533|     //
71534|     // Begin by determine whether or not the file
    | system is a disk-based file
71535|     // system. If not, then this driver is not
    | concerned with it.
71536|     //
71537|
71538|     if ( DeviceObject->DeviceType !=
    | FILE_DEVICE_DISK_FILE_SYSTEM ) {
71539|         Debug(DEBUG_SFILTER,("SFILTER: Not a disk file
    | system!\n"));
71540|         return;
71541|     }
71542|
71543|     //
71544|     // Begin by determining whether this file system is
    | registering or
71545|     // unregistering as an active file system.
71546|     //
71547|
71548|     if ( FsActive ) {
71549|

```

```

71550|     PFS_FILTER_EXTENSION deviceExtension;
71551|
71552|     //
71553|     // The file system has registered as an active
       | file system. If it is
71554|     // a disk-based file system attach to it.
71555|     //
71556|
71557|     FsRtlEnterFileSystem();
71558|     ExAcquireResourceExclusive( &FsLock, TRUE );
71559| #ifdef DEBUG_EXTENSION
71560|     ULONG SizeOfDeviceExt =
       | sizeof(DEVICE_EXTENSION);
71561| #else
71562|     ULONG SizeOfDeviceExt = sizeof(
       | FS_FILTER_EXTENSION);
71563| #endif
71564|     status = IoCreateDevice(
71565|         PSMAN_DRIVER_OBJECT,
71566|         SizeOfDeviceExt,
71567|         (PUNICODE_STRING) NULL,
71568|         FILE_DEVICE_DISK_FILE_SYSTEM,
71569|         0,
71570|         FALSE,
71571|         &deviceObject
71572|     );
71573|
71574|     if ( NT_SUCCESS( status ) ) {
71575| #ifdef DEBUG_EXTENSION
71576|         | ((PDEVICE_EXTENSION)(deviceObject->DeviceExtension))->Ob
       | jectType = OBJECT_FS_FILTER;
71577|         | ((PDEVICE_EXTENSION)(deviceObject->DeviceExtension))->Re
       | alDeviceExtension =
       | MemAllocatePoolWithTag(NonPagedPool,FS_FILTER_EXTENSION_
       | SIZE,DEVEXTTAG);
71578|         | RtlZeroMemory(((PDEVICE_EXTENSION)(deviceObject->DeviceE
       | xtension))->RealDeviceExtension,FS_FILTER_EXTENSION_SIZE
       | );
71579| #endif
71580|
71581|         #ifdef DO_ALL_SFILTER
71582|         Debug(DEBUG_SFILTER,("SFILTER: created
       | filter %08x for filesystem
       | %08x\n",deviceObject,DeviceObject));
71583|         #endif /*DO_ALL_SFILTER*/
71584|

```

```

71585|         deviceExtension =
| (PFS_FILTER_EXTENSION)GetDeviceExtension(deviceObject);
71586|         deviceExtension->DeviceObject =
| deviceObject;
71587|         deviceExtension->DriverObject =
| PSMANDriverObject;
71588|         deviceExtension->ObjectType =
| OBJECT_FS_FILTER;
71589|         deviceExtension->Attached = FALSE;
71590|         deviceExtension->TargetDeviceObject = NULL;
71591|         deviceExtension->PSMStorageObject = NULL;
71592|         deviceExtension->Virtual = FALSE;
71593|         deviceExtension->FileSystem = TRUE;
71594|         UNICODE_STRING Check;
71595|
| RtlInitUnicodeString(&Check,L"\\FileSystem\\RAW");
71596|
| if(RtlCompareUnicodeString(&Check,&DeviceObject->DriverO
| bject->DriverName,TRUE)==0) {
71597|         deviceExtension->IsRaw=TRUE;
71598|     }
71599|
| RtlInitUnicodeString(&Check,L"\\FileSystem\\Fastfat");
71600|
| if(RtlCompareUnicodeString(&Check,&DeviceObject->DriverO
| bject->DriverName,TRUE)==0) {
71601|         deviceExtension->IsFat=TRUE;
71602|     }
71603|
| RtlInitUnicodeString(&Check,L"\\FileSystem\\Ntfs");
71604|
| if(RtlCompareUnicodeString(&Check,&DeviceObject->DriverO
| bject->DriverName,TRUE)==0) {
71605|         deviceExtension->IsNtfs=TRUE;
71606|     }
71607|
DeviceObject = IoAttachDeviceToDeviceStack(
| deviceObject, DeviceObject );
71609|     if ( DeviceObject == NULL ) {
71610|         Debug(DEBUG_SFILTER,("SFILTER: Unable
| to create fs filter\n"));
71611|         IoDeleteDevice( deviceObject );
71612|     } else {
71613|         deviceExtension->TargetDeviceObject =
| DeviceObject;
71614|         deviceExtension->Attached = TRUE;
71615|         deviceObject->Flags &=
| ~DO_DEVICE_INITIALIZING;
71616|     }
71617| }

```

```

71618|     ExReleaseResource( &FsLock );
71619|     FsRtlExitFileSystem();
71620| } else {
71621|
71622|     //
71623|     // Search the linked list of drivers attached
71624|     | to this device and check
71625|     // to see whether this driver is attached to
71626|     | it. If so, remove it.
71627|     //
71628|     if ( nextAttachedDevice =
71629|         | DeviceObject->AttachedDevice ) {
71630|
71631|         PFS_FILTER_EXTENSION deviceExtension;
71632|
71633|         // This registered file system has someone
71634|         | attached to it. Scan
71635|         // until this driver's device object is
71636|         | found and detach it.
71637|         //
71638|         FsRtlEnterFileSystem();
71639|         ExAcquireResourceShared( &FsLock, TRUE );
71640|
71641|         while ( nextAttachedDevice ) {
71642|             deviceExtension =
71643|             | (PFS_FILTER_EXTENSION)GetDeviceExtension(nextAttachedDev
71644|             | ice);
71645|             if (
71646|             | PsmGetObjectType(nextAttachedDevice)==OBJECT_FS_FILTER
71647|             | ) {
71648|
71649|                 //
71650|                 // A device object that may belong
71651|                 | to this driver has been
71652|                 // located. Scan the list of
71653|                 | device objects owned by this
71654|                 // driver to see whether or not is
71655|                 | actually belongs to this
71656|                 // driver.
71657|                 //
71658|                 fsDevice =
71659|                 | PSMAN_DRIVER_OBJECT->DeviceObject;
71660|                 while ( fsDevice ) {
71661|                     if ( fsDevice ==
71662|                     | nextAttachedDevice ) {
71663|
71664|                         ASSERT(PsmGetObjectType(fsDevice)==OBJECT_FS_FILTER);

```

```

71653|
    | Debug(DEBUG_SFILTER,("SFILTER: Detaching from
    | %08x\n",DeviceObject));
71654|                IoDetachDevice(
    | DeviceObject );
71655|                deviceExtension =
    | (PFS_FILTER_EXTENSION)GetDeviceExtension(fsDevice);
71656|                deviceExtension->Attached =
    | FALSE;
71657|                if (
    | !fsDevice->AttachedDevice ) {
71658|                IoDeleteDevice(
    | fsDevice );
71659|                }
71660|                // **** What to do if still
    | attached?
71661|                ExReleaseResource( &FsLock
    | );
71662|                FsRtlExitFileSystem();
71663|                return;
71664|                }
71665|                fsDevice =
    | fsDevice->NextDevice;
71666|                }
71667|                }
71668|                DeviceObject = nextAttachedDevice;
71669|                nextAttachedDevice =
    | nextAttachedDevice->AttachedDevice;
71670|                }
71671|                ExReleaseResource( &FsLock );
71672|                FsRtlExitFileSystem();
71673|                }
71674|        }
71675|
71676|    return;
71677| }
71678|
71679|
71680| // returns true if it can handle the request, otherwise
    | false
71681| STATIC
71682| BOOLEAN
71683| FastIoCommonStuff( PDEVICE_OBJECT DeviceObject,
    | PFILE_OBJECT FileObject, char *Where )
71684| {
71685|     PAGED_CODE();
71686|
71687|     if (
    | PsmGetObjectType(DeviceObject)!=OBJECT_FS_FILTER ) {
71688|         Debug(DEBUG_SFILTER,("SFILTER: %s: Not fs

```

```

    | object\n",Where));
71689|     return TRUE;
71690| }
71691|
71692|
    | ASSERT(PsmGetObjectype(DeviceObject)==OBJECT_FS_FILTER)
    | ;
71693|
71694|     return FALSE;
71695| }
71696|
71697|
71698| STATIC
71699| BOOLEAN
71700| SfFastIoCheckIfPossible(
71701|     IN PFILE_OBJECT FileObject,
71702|     IN PLARGE_INTEGER FileOffset,
71703|     IN ULONG Length,
71704|     IN BOOLEAN Wait,
71705|     IN ULONG LockKey,
71706|     IN BOOLEAN
    | CheckForReadOperation,
71707|     OUT PIO_STATUS_BLOCK IoStatus,
71708|     IN PDEVICE_OBJECT DeviceObject
71709| )
71710|
71711| /*++
71712|
71713| Routine Description:
71714|
71715| This routine is the fast I/O "pass through" routine
    | for checking to see
71716| whether fast I/O is possible for this file.
71717|
71718| This function simply invokes the file system's
    | cooresponding routine, or
71719| returns FALSE if the file system does not implement
    | the function.
71720|
71721| Arguments:
71722|
71723| FileObject - Pointer to the file object to be
    | operated on.
71724|
71725| FileOffset - Byte offset in the file for the
    | operation.
71726|
71727| Length - Length of the operation to be performed.
71728|
71729| Wait - Indicates whether or not the caller is

```

```

    | willing to wait if the
71730|     appropriate locks, etc. cannot be acquired
71731|
71732|     LockKey - Provides the caller's key for file locks.
71733|
71734|     CheckForReadOperation - Indicates whether the
    | caller is checking for a
71735|         read (TRUE) or a write operation.
71736|
71737|     IoStatus - Pointer to a variable to receive the I/O
    | status of the
71738|         operation.
71739|
71740|     DeviceObject - Pointer to this driver's device
    | object, the device on
71741|         which the operation is to occur.
71742|
71743| Return Value:
71744|
71745|     The function value is TRUE or FALSE based on
    | whether or not fast I/O
71746|         is possible for this file.
71747|
71748| --*/
71749|
71750| {
71751|     PDEVICE_OBJECT deviceObject;
71752|     PFAST_IO_DISPATCH fastIoDispatch;
71753|
71754|
    | if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoChec
    | kIfIoPossible")) {
71755|         return FALSE;
71756|     }
71757|
71758|     deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
71759|     if ( !deviceObject ) {
71760|         return FALSE;
71761|     }
71762|
71763|     fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
71764|
71765|     if ( fastIoDispatch &&
    | fastIoDispatch->FastIoCheckIfPossible ) {
71766|         return(fastIoDispatch->FastIoCheckIfPossible)(
71767|             | FileObject,
71768|

```

```

    | FileOffset,
71769|
    | Length,
71770|
    | Wait,
71771|
    | LockKey,
71772|
    | CheckForReadOperation,
71773|
    | IoStatus,
71774|
    | deviceObject
71775|                                     );
71776|     } else {
71777|         return FALSE;
71778|     }
71779|
71780| }
71781|
71782| STATIC
71783| BOOLEAN
71784| SfFastIoRead(
71785|     IN PFILE_OBJECT FileObject,
71786|     IN PLARGE_INTEGER FileOffset,
71787|     IN ULONG Length,
71788|     IN BOOLEAN Wait,
71789|     IN ULONG LockKey,
71790|     OUT PVOID Buffer,
71791|     OUT PIO_STATUS_BLOCK IoStatus,
71792|     IN PDEVICE_OBJECT DeviceObject
71793| )
71794|
71795| /*++
71796|
71797| Routine Description:
71798|
71799|     This routine is the fast I/O "pass through" routine
    | for reading from a
71800|     file.
71801|
71802|     This function simply invokes the file system's
    | corresponding routine, or
71803|     returns FALSE if the file system does not implement
    | the function.
71804|
71805| Arguments:
71806|
71807|     FileObject - Pointer to the file object to be read.
71808|

```



```

71809|   FileOffset - Byte offset in the file of the read.
71810|
71811|   Length - Length of the read operation to be
       | performed.
71812|
71813|   Wait - Indicates whether or not the caller is
       | willing to wait if the
71814|         appropriate locks, etc. cannot be acquired
71815|
71816|   LockKey - Provides the caller's key for file locks.
71817|
71818|   Buffer - Pointer to the caller's buffer to receive
       | the data read.
71819|
71820|   IoStatus - Pointer to a variable to receive the I/O
       | status of the
71821|         operation.
71822|
71823|   DeviceObject - Pointer to this driver's device
       | object, the device on
71824|         which the operation is to occur.
71825|
71826| Return Value:
71827|
71828|   The function value is TRUE or FALSE based on
       | whether or not fast I/O
71829|         is possible for this file.
71830|
71831| --*/
71832|
71833| {
71834|   PDEVICE_OBJECT deviceObject;
71835|   PFAST_IO_DISPATCH fastIoDispatch;
71836|
71837|   | if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoRead
       | ")) {
71838|       return FALSE;
71839|   }
71840|
71841|   deviceObject = ((PFS_FILTER_EXTENSION)
       | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
71842|   if ( !deviceObject ) {
71843|       return FALSE;
71844|   }
71845|   fastIoDispatch =
       | deviceObject->DriverObject->FastIoDispatch;
71846|
71847|   if ( fastIoDispatch && fastIoDispatch->FastIoRead )
       | {

```

```

71848|     return(fastIoDispatch->FastIoRead)(
71849|                                     FileObject,
71850|                                     FileOffset,
71851|                                     Length,
71852|                                     Wait,
71853|                                     LockKey,
71854|                                     Buffer,
71855|                                     IoStatus,
71856|                                     deviceObject
71857|                                     );
71858| } else {
71859|     return FALSE;
71860| }
71861| }
71862|
71863| STATIC
71864| BOOLEAN
71865| SfFastIoWrite(
71866|     IN PFILE_OBJECT FileObject,
71867|     IN PLARGE_INTEGER FileOffset,
71868|     IN ULONG Length,
71869|     IN BOOLEAN Wait,
71870|     IN ULONG LockKey,
71871|     IN PVOID Buffer,
71872|     OUT PIO_STATUS_BLOCK IoStatus,
71873|     IN PDEVICE_OBJECT DeviceObject
71874|     )
71875|
71876| /*++
71877|
71878| Routine Description:
71879|
71880| This routine is the fast I/O "pass through" routine
71881| | for writing to a
71882| | file.
71883|
71884| This function simply invokes the file system's
71885| | cooresponding routine, or
71886| returns FALSE if the file system does not implement
71887| | the function.
71888|
71889| Arguments:
71890|
71891| FileObject - Pointer to the file object to be
71892| | written.
71893|
71894| FileOffset - Byte offset in the file of the write
71895| | operation.
71896|
71897| Length - Length of the write operation to be

```

```

    | performed.
71893|
71894|    Wait - Indicates whether or not the caller is
    | willing to wait if the
71895|        appropriate locks, etc. cannot be acquired
71896|
71897|    LockKey - Provides the caller's key for file locks.
71898|
71899|    Buffer - Pointer to the caller's buffer that
    | contains the data to be
71900|        written.
71901|
71902|    IoStatus - Pointer to a variable to receive the I/O
    | status of the
71903|        operation.
71904|
71905|    DeviceObject - Pointer to this driver's device
    | object, the device on
71906|        which the operation is to occur.
71907|
71908| Return Value:
71909|
71910|    The function value is TRUE or FALSE based on
    | whether or not fast I/O
71911|        is possible for this file.
71912|
71913| --*/
71914|
71915| {
71916|     PDEVICE_OBJECT deviceObject;
71917|     PFAST_IO_DISPATCH fastIoDispatch;
71918|     PFS_FILTER_EXTENSION
    | DevExt=(PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceOb
    | ject);
71919|
71920|
    | if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoWrit
    | e")) {
71921|         return FALSE;
71922|     }
71923|
71924|     if ( (DevExt->Virtual) &&
    | (DevExt->PSMStorageObject) ) {
71925|         PVDISK_EXTENSION
    | VDiskExt=(PVDISK_EXTENSION)GetDeviceExtension(DevExt->PS
    | MStorageObject);
71926|
71927|         if ( VDiskExt->SnapShot ) {
71928|             pPersistentDictionary dictionary =
    | (pPersistentDictionary) VDiskExt->SnapShot->Dictionary;

```

```

71929|      ASSERT ( dictionary != NULL );
71930|      if ( dictionary != NULL ) {
71931|          if ( dictionary->IsReadOnly() ) {
71932|              if ( FileIsReadOnly(FileObject) ) {
71933|                  // virtual volume is marked
71934|                  | readonly, deny this write
71935|                  Debug(DEBUG_SFILTER,("SFILTER:
71936|                  | FastIoWrite: Write to readonly volume\n"));
71937|                  IoStatus->Information = 0;
71938|                  IoStatus->Status =
71939|                  | STATUS_MEDIA_WRITE_PROTECTED;
71940|                  return TRUE;
71941|              }
71942|          } else {
71943|              // not a read-only snapshot. Need
71944|              | to check cache usage...
71945|              if ( !(gVDiskIOHandling &
71946|              | PSM_VDISK_FLAG_CACHE_FULL_DELETE_SS) ) {
71947|                  if (
71948|                  | dictionary->IsCacheWarningThresholdReached() ) {
71949|                      if (
71950|                      | FileIsOpenOnSnapShot(FileObject,FALSE) ) {
71951|                          // Fail the write to
71952|                          | the snapshot because too much cache is in use...
71953|                          NTSTATUS Status =
71954|                          | IoStatus->Status = STATUS_DISK_FULL;
71955|                          IoStatus->Information =
71956|                          | 0;
71957|                          | Debug(DEBUG_SFILTER,("SfFastIoWrite: Reporting
71958|                          | STATUS_DISK_FULL; DevExt=%08x\n",DevExt));
71959|                          return TRUE;
71960|                      }
71961|                  }
71962|              }
71963|          }
71964|      }
71965|      }
71966|      }
71967|      }
71968|      }
71969|      }
71970|      }
71971|      }
71972|      }
71973|      }
71974|      }
71975|      }
71976|      }
71977|      }
71978|      }
71979|      }
71980|      }
71981|      }
71982|      }
71983|      }
71984|      }
71985|      }
71986|      }
71987|      }
71988|      }
71989|      }
71990|      }
71991|      }
71992|      }
71993|      }
71994|      }
71995|      }
71996|      }
71997|      }
71998|      }
71999|      }
72000|      }
72001|      }
72002|      }
72003|      }
72004|      }
72005|      }
72006|      }
72007|      }
72008|      }
72009|      }
72010|      }
72011|      }
72012|      }
72013|      }
72014|      }
72015|      }
72016|      }
72017|      }
72018|      }
72019|      }
72020|      }
72021|      }
72022|      }
72023|      }
72024|      }
72025|      }
72026|      }
72027|      }
72028|      }
72029|      }
72030|      }
72031|      }
72032|      }
72033|      }
72034|      }
72035|      }
72036|      }
72037|      }
72038|      }
72039|      }
72040|      }
72041|      }
72042|      }
72043|      }
72044|      }
72045|      }
72046|      }
72047|      }
72048|      }
72049|      }
72050|      }
72051|      }
72052|      }
72053|      }
72054|      }
72055|      }
72056|      }
72057|      }
72058|      }
72059|      }
72060|      }
72061|      }
72062|      }
72063|      }
72064|      }
72065|      }
72066|      }
72067|      }
72068|      }
72069|      }
72070|      }
72071|      }
72072|      }
72073|      }
72074|      }
72075|      }
72076|      }
72077|      }
72078|      }
72079|      }
72080|      }
72081|      }
72082|      }
72083|      }
72084|      }
72085|      }
72086|      }
72087|      }
72088|      }
72089|      }
72090|      }
72091|      }
72092|      }
72093|      }
72094|      }
72095|      }
72096|      }
72097|      }
72098|      }
72099|      }
72100|      }
72101|      }
72102|      }
72103|      }
72104|      }
72105|      }
72106|      }
72107|      }
72108|      }
72109|      }
72110|      }
72111|      }
72112|      }
72113|      }
72114|      }
72115|      }
72116|      }
72117|      }
72118|      }
72119|      }
72120|      }
72121|      }
72122|      }
72123|      }
72124|      }
72125|      }
72126|      }
72127|      }
72128|      }
72129|      }
72130|      }
72131|      }
72132|      }
72133|      }
72134|      }
72135|      }
72136|      }
72137|      }
72138|      }
72139|      }
72140|      }
72141|      }
72142|      }
72143|      }
72144|      }
72145|      }
72146|      }
72147|      }
72148|      }
72149|      }
72150|      }
72151|      }
72152|      }
72153|      }
72154|      }
72155|      }
72156|      }
72157|      }
72158|      }
72159|      }
72160|      }
72161|      }
72162|      }
72163|      }
72164|      }
72165|      }
72166|      }
72167|      }
72168|      }
72169|      }
72170|      }
72171|      }
72172|      }
72173|      }
72174|      }
72175|      }
72176|      }
72177|      }
72178|      }
72179|      }
72180|      }
72181|      }
72182|      }
72183|      }
72184|      }
72185|      }
72186|      }
72187|      }
72188|      }
72189|      }
72190|      }
72191|      }
72192|      }
72193|      }
72194|      }
72195|      }
72196|      }
72197|      }
72198|      }
72199|      }
72200|      }
72201|      }
72202|      }
72203|      }
72204|      }
72205|      }
72206|      }
72207|      }
72208|      }
72209|      }
72210|      }
72211|      }
72212|      }
72213|      }
72214|      }
72215|      }
72216|      }
72217|      }
72218|      }
72219|      }
72220|      }
72221|      }
72222|      }
72223|      }
72224|      }
72225|      }
72226|      }
72227|      }
72228|      }
72229|      }
72230|      }
72231|      }
72232|      }
72233|      }
72234|      }
72235|      }
72236|      }
72237|      }
72238|      }
72239|      }
72240|      }
72241|      }
72242|      }
72243|      }
72244|      }
72245|      }
72246|      }
72247|      }
72248|      }
72249|      }
72250|      }
72251|      }
72252|      }
72253|      }
72254|      }
72255|      }
72256|      }
72257|      }
72258|      }
72259|      }
72260|      }
72261|      }
72262|      }
72263|      }
72264|      }
72265|      }
72266|      }
72267|      }
72268|      }
72269|      }
72270|      }
72271|      }
72272|      }
72273|      }
72274|      }
72275|      }
72276|      }
72277|      }
72278|      }
72279|      }
72280|      }
72281|      }
72282|      }
72283|      }
72284|      }
72285|      }
72286|      }
72287|      }
72288|      }
72289|      }
72290|      }
72291|      }
72292|      }
72293|      }
72294|      }
72295|      }
72296|      }
72297|      }
72298|      }
72299|      }
72300|      }
72301|      }
72302|      }
72303|      }
72304|      }
72305|      }
72306|      }
72307|      }
72308|      }
72309|      }
72310|      }
72311|      }
72312|      }
72313|      }
72314|      }
72315|      }
72316|      }
72317|      }
72318|      }
72319|      }
72320|      }
72321|      }
72322|      }
72323|      }
72324|      }
72325|      }
72326|      }
72327|      }
72328|      }
72329|      }
72330|      }
72331|      }
72332|      }
72333|      }
72334|      }
72335|      }
72336|      }
72337|      }
72338|      }
72339|      }
72340|      }
72341|      }
72342|      }
72343|      }
72344|      }
72345|      }
72346|      }
72347|      }
72348|      }
72349|      }
72350|      }
72351|      }
72352|      }
72353|      }
72354|      }
72355|      }
72356|      }
72357|      }
72358|      }
72359|      }
723
```

```

71964|     return(fastIoDispatch->FastIoWrite)(
71965|                                     FileObject,
71966|                                     FileOffset,
71967|                                     Length,
71968|                                     Wait,
71969|                                     LockKey,
71970|                                     Buffer,
71971|                                     IoStatus,
71972|                                     deviceObject
71973|                                     );
71974| } else {
71975|     return FALSE;
71976| }
71977| }
71978|
71979| STATIC
71980| BOOLEAN
71981| SfFastIoQueryBasicInfo(
71982|     IN PFILE_OBJECT FileObject,
71983|     IN BOOLEAN Wait,
71984|     OUT PFILE_BASIC_INFORMATION
71985|     | Buffer,
71986|     OUT PIO_STATUS_BLOCK IoStatus,
71987|     IN PDEVICE_OBJECT DeviceObject
71988|     )
71989| /*++
71990|
71991| Routine Description:
71992|
71993| This routine is the fast I/O "pass through" routine
71994| | for querying basic
71995| information about the file.
71996|
71997| This function simply invokes the file system's
71998| | corresponding routine, or
71999| returns FALSE if the file system does not implement
72000| | the function.
72001|
72002| Arguments:
72003|
72004| FileObject - Pointer to the file object to be
72005| | queried.
72006|
72007| Wait - Indicates whether or not the caller is
72008| | willing to wait if the
72009| appropriate locks, etc. cannot be acquired
72010|
72011| Buffer - Pointer to the caller's buffer to receive
72012| | the information about

```

```

72007|         the file.
72008|
72009|     IoStatus - Pointer to a variable to receive the I/O
        | status of the
72010|         operation.
72011|
72012|     DeviceObject - Pointer to this driver's device
        | object, the device on
72013|         which the operation is to occur.
72014|
72015| Return Value:
72016|
72017|     The function value is TRUE or FALSE based on
        | whether or not fast I/O
72018|     is possible for this file.
72019|
72020| --*/
72021|
72022| {
72023|     PDEVICE_OBJECT deviceObject;
72024|     PFAST_IO_DISPATCH fastIoDispatch;
72025|
72026|     if (FastIoCommonStuff(DeviceObject, FileObject, "FastIoQuer
        | yBasicInfo")) {
72027|         return FALSE;
72028|     }
72029|
72030|     deviceObject = ((PFS_FILTER_EXTENSION)
        | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72031|     if ( !deviceObject ) {
72032|         return FALSE;
72033|     }
72034|     fastIoDispatch =
        | deviceObject->DriverObject->FastIoDispatch;
72035|
72036|     if ( fastIoDispatch &&
        | fastIoDispatch->FastIoQueryBasicInfo ) {
72037|         return(fastIoDispatch->FastIoQueryBasicInfo)(
72038|             | FileObject,
72039|             | Wait,
72040|             | Buffer,
72041|             | IoStatus,
72042|             | deviceObject
72043|             );

```

```

72044|    } else {
72045|        return FALSE;
72046|    }
72047| }
72048|
72049| STATIC
72050| BOOLEAN
72051| SfFastIoQueryStandardInfo(
72052|     IN PFILE_OBJECT FileObject,
72053|     IN BOOLEAN Wait,
72054|     OUT PFILE_STANDARD_INFORMATION
72055|     | Buffer,
72056|     OUT PIO_STATUS_BLOCK IoStatus,
72057|     IN PDEVICE_OBJECT DeviceObject
72058| )
72059| /*++
72060|
72061| Routine Description:
72062|
72063| This routine is the fast I/O "pass through" routine
72064| | for querying standard
72065| information about the file.
72066|
72067| This function simply invokes the file system's
72068| | cooresponding routine, or
72069| returns FALSE if the file system does not implement
72070| | the function.
72071|
72072| Arguments:
72073|
72074| FileObject - Pointer to the file object to be
72075| | queried.
72076|
72077| Wait - Indicates whether or not the caller is
72078| | willing to wait if the
72079| appropriate locks, etc. cannot be acquired
72080|
72081| Buffer - Pointer to the caller's buffer to receive
72082| | the information about
72083| the file.
72084|
72085| IoStatus - Pointer to a variable to receive the I/O
72086| | status of the
72087| operation.
72088|
72089| DeviceObject - Pointer to this driver's device
72090| | object, the device on
72091| which the operation is to occur.
72092|
72093|
72094|

```

```

72085| Return Value:
72086|
72087|   The function value is TRUE or FALSE based on
       | whether or not fast I/O
72088|   is possible for this file.
72089|
72090| --*/
72091|
72092| {
72093|   PDEVICE_OBJECT deviceObject;
72094|   PFAST_IO_DISPATCH fastIoDispatch;
72095|
72096|   if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoQuer
       | yStandardInfo")) {
72097|       return FALSE;
72098|   }
72099|
72100|   deviceObject = ((PFS_FILTER_EXTENSION)
       | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72101|   if ( !deviceObject ) {
72102|       return FALSE;
72103|   }
72104|   fastIoDispatch =
       | deviceObject->DriverObject->FastIoDispatch;
72105|
72106|   if ( fastIoDispatch &&
       | fastIoDispatch->FastIoQueryStandardInfo ) {
72107|       return(fastIoDispatch->FastIoQueryStandardInfo)(
72108|       | FileObject,
72109|       | Wait,
72110|       | Buffer,
72111|       | IoStatus,
72112|       | deviceObject
72113|       | );
72114|   } else {
72115|       return FALSE;
72116|   }
72117| }
72118|
72119| STATIC
72120| BOOLEAN
72121| SfFastIoLock(

```



```

72122|      IN PFILE_OBJECT FileObject,
72123|      IN PLARGE_INTEGER FileOffset,
72124|      IN PLARGE_INTEGER Length,
72125|      PEPROCESS ProcessId,
72126|      ULONG Key,
72127|      BOOLEAN FailImmediately,
72128|      BOOLEAN ExclusiveLock,
72129|      OUT PIO_STATUS_BLOCK IoStatus,
72130|      IN PDEVICE_OBJECT DeviceObject
72131|  )
72132|
72133| /*++
72134|
72135| Routine Description:
72136|
72137|   This routine is the fast I/O "pass through" routine
72138|   | for locking a byte
72139|   | range within a file.
72140|
72141|   This function simply invokes the file system's
72142|   | corresponding routine, or
72143|   | returns FALSE if the file system does not implement
72144|   | the function.
72145|
72146| Arguments:
72147|
72148|   FileObject - Pointer to the file object to be
72149|   | locked.
72150|
72151|   FileOffset - Starting byte offset from the base of
72152|   | the file to be locked.
72153|
72154|   Length - Length of the byte range to be locked.
72155|
72156|   ProcessId - ID of the process requesting the file
72157|   | lock.
72158|
72159|   Key - Lock key to associate with the file lock.
72160|
72161|   FailImmediately - Indicates whether or not the lock
72162|   | request is to fail
72163|   | if it cannot be immediately be granted.
72164|
72165|   ExclusiveLock - Indicates whether the lock to be
72166|   | taken is exclusive (TRUE)
72167|   | or shared.
72168|
72169|   IoStatus - Pointer to a variable to receive the I/O
72170|   | status of the
72171|   | operation.

```

```

72163|
72164| DeviceObject - Pointer to this driver's device
| object, the device on
72165| which the operation is to occur.
72166|
72167| Return Value:
72168|
72169| The function value is TRUE or FALSE based on
| whether or not fast I/O
72170| is possible for this file.
72171|
72172| --*/
72173|
72174| {
72175| PDEVICE_OBJECT deviceObject;
72176| PFAST_IO_DISPATCH fastIoDispatch;
72177|
72178|
| if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoLock
| ")) {
72179| return FALSE;
72180| }
72181|
72182| deviceObject = ((PFS_FILTER_EXTENSION)
| (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72183| if ( !deviceObject ) {
72184| return FALSE;
72185| }
72186| fastIoDispatch =
| deviceObject->DriverObject->FastIoDispatch;
72187|
72188| if ( fastIoDispatch && fastIoDispatch->FastIoLock )
| {
72189| return(fastIoDispatch->FastIoLock)(
72190| FileObject,
72191| FileOffset,
72192| Length,
72193| ProcessId,
72194| Key,
72195|
| Faillmmediately,
72196|
| ExclusiveLock,
72197| IoStatus,
72198| deviceObject
72199| );
72200| } else {
72201| return FALSE;
72202| }
72203|

```

```

72204| }
72205|
72206| STATIC
72207| BOOLEAN
72208| SfFastIoUnlockSingle(
72209|         IN PFILE_OBJECT FileObject,
72210|         IN PLARGE_INTEGER FileOffset,
72211|         IN PLARGE_INTEGER Length,
72212|         PEPROCESS ProcessId,
72213|         ULONG Key,
72214|         OUT PIO_STATUS_BLOCK IoStatus,
72215|         IN PDEVICE_OBJECT DeviceObject
72216|     )
72217|
72218| /*++
72219|
72220| Routine Description:
72221|
72222|   This routine is the fast I/O "pass through" routine
72223|   | for unlocking a byte
72224|   | range within a file.
72225|
72226|   This function simply invokes the file system's
72227|   | cooresponding routine, or
72228|   | returns FALSE if the file system does not implement
72229|   | the function.
72230|
72231| Arguments:
72232|
72233|   FileObject - Pointer to the file object to be
72234|   | unlocked.
72235|
72236|   FileOffset - Starting byte offset from the base of
72237|   | the file to be
72238|   | unlocked.
72239|
72240|   Length - Length of the byte range to be unlocked.
72241|
72242|   ProcessId - ID of the process requesting the unlock
72243|   | operation.
72244|
72245|   Key - Lock key associated with the file lock.
72246|
72247|   IoStatus - Pointer to a variable to receive the I/O
72248|   | status of the
72249|   | operation.
72250|
72251|   DeviceObject - Pointer to this driver's device
72252|   | object, the device on
72253|   | which the operation is to occur.

```

```

72246|
72247| Return Value:
72248|
72249| The function value is TRUE or FALSE based on
    | whether or not fast I/O
72250| is possible for this file.
72251|
72252| --*/
72253|
72254| {
72255|     PDEVICE_OBJECT deviceObject;
72256|     PFAST_IO_DISPATCH fastIoDispatch;
72257|
72258|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoUnlo
    | ckSingle")) {
72259|         return FALSE;
72260|     }
72261|
72262|     deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72263|     if ( !deviceObject ) {
72264|         return FALSE;
72265|     }
72266|     fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
72267|
72268|     if ( fastIoDispatch &&
    | fastIoDispatch->FastIoUnlockSingle ) {
72269|         return(fastIoDispatch->FastIoUnlockSingle)(
72270|             | FileObject,
72271|             | FileOffset,
72272|             | Length,
72273|             | ProcessId,
72274|             | Key,
72275|             | IoStatus,
72276|             | deviceObject
    | );
72277|     } else {
72278|         return FALSE;
72279|     }
72280| }
72281| }
72282|
72283| STATIC

```

```

72284| BOOLEAN
72285| SfFastIoUnlockAll(
72286|         IN PFILE_OBJECT FileObject,
72287|         PEPROCESS ProcessId,
72288|         OUT PIO_STATUS_BLOCK IoStatus,
72289|         IN PDEVICE_OBJECT DeviceObject
72290|         )
72291|
72292| /*++
72293|
72294| Routine Description:
72295|
72296|   This routine is the fast I/O "pass through" routine
72297|   | for unlocking all
72298|   locks within a file.
72299|
72300|   This function simply invokes the file system's
72301|   | cooresponding routine, or
72302|   returns FALSE if the file system does not implement
72303|   | the function.
72304|
72305| Arguments:
72306|
72307|   FileObject - Pointer to the file object to be
72308|   | unlocked.
72309|
72310|   ProcessId - ID of the process requesting the unlock
72311|   | operation.
72312|
72313|   IoStatus - Pointer to a variable to receive the I/O
72314|   | status of the
72315|   operation.
72316|
72317|   DeviceObject - Pointer to this driver's device
72318|   | object, the device on
72319|   which the operation is to occur.
72320|
72321| Return Value:
72322|
72323|   The function value is TRUE or FALSE based on
72324|   | whether or not fast I/O
72325|   is possible for this file.
72326|
72327| --*/
72328| {
72329|     PDEVICE_OBJECT deviceObject;
72330|     PFAST_IO_DISPATCH fastIoDispatch;
72331|
72332|     if (FileObject == NULL)
72333|         return FALSE;
72334|
72335|     if (!PEPROCESS_VALID(ProcessId))
72336|         return FALSE;
72337|
72338|     if (!PDEVICE_OBJECT_VALID(DeviceObject))
72339|         return FALSE;
72340|
72341|     deviceObject = DeviceObject;
72342|     fastIoDispatch = DeviceObject->FastIoDispatch;
72343|
72344|     if (!fastIoDispatch->SfFastIoUnlockAll)
72345|         return FALSE;
72346|
72347|     return fastIoDispatch->SfFastIoUnlockAll(
72348|         FileObject,
72349|         ProcessId,
72350|         IoStatus,
72351|         DeviceObject);
72352| }

```

```

    | if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoUnlo
    | ckAll")) {
72326|         return FALSE;
72327|     }
72328|
72329|     deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72330|     if ( !deviceObject ) {
72331|         return FALSE;
72332|     }
72333|     fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
72334|
72335|     if ( fastIoDispatch &&
    | fastIoDispatch->FastIoUnlockAll ) {
72336|         return(fastIoDispatch->FastIoUnlockAll)(
72337|
    | FileObject,
72338|
    | ProcessId,
72339|
    | IoStatus,
72340|
    | deviceObject
72341|                                     );
72342|     } else {
72343|         return FALSE;
72344|     }
72345| }
72346|
72347| STATIC
72348| BOOLEAN
72349| SfFastIoUnlockAllByKey(
72350|         IN PFILE_OBJECT FileObject,
72351|         PVOID ProcessId,
72352|         ULONG Key,
72353|         OUT PIO_STATUS_BLOCK IoStatus,
72354|         IN PDEVICE_OBJECT DeviceObject
72355|         )
72356|
72357| /*++
72358|
72359| Routine Description:
72360|
72361| This routine is the fast I/O "pass through" routine
    | for unlocking all
72362| locks within a file based on a specified key.
72363|
72364| This function simply invokes the file system's
    | cooresponding routine, or

```

```

72365|    returns FALSE if the file system does not implement
      | the function.
72366|
72367| Arguments:
72368|
72369|    FileObject - Pointer to the file object to be
      | unlocked.
72370|
72371|    ProcessId - ID of the process requesting the unlock
      | operation.
72372|
72373|    Key - Lock key associated with the locks on the
      | file to be released.
72374|
72375|    IoStatus - Pointer to a variable to receive the I/O
      | status of the
72376|    operation.
72377|
72378|    DeviceObject - Pointer to this driver's device
      | object, the device on
72379|    which the operation is to occur.
72380|
72381| Return Value:
72382|
72383|    The function value is TRUE or FALSE based on
      | whether or not fast I/O
72384|    is possible for this file.
72385|
72386| --*/
72387|
72388| {
72389|     PDEVICE_OBJECT deviceObject;
72390|     PFAST_IO_DISPATCH fastIoDispatch;
72391|
72392|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoUnlo
      | ckAllByKey")) {
72393|         return FALSE;
72394|     }
72395|
72396|     deviceObject = ((PFS_FILTER_EXTENSION)
      | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72397|     if ( !deviceObject ) {
72398|         return FALSE;
72399|     }
72400|     fastIoDispatch =
      | deviceObject->DriverObject->FastIoDispatch;
72401|
72402|     if ( fastIoDispatch &&
      | fastIoDispatch->FastIoUnlockAllByKey ) {

```

```

72403|     return(fastIoDispatch->FastIoUnlockAllByKey)(
72404|         | FileObject,
72405|         | ProcessId,
72406|         | Key,
72407|         | IoStatus,
72408|         | deviceObject
72409|         );
72410|     } else {
72411|         return FALSE;
72412|     }
72413|
72414| }
72415|
72416| STATIC
72417| BOOLEAN
72418| SfFastIoDeviceControl(
72419|     IN PFILE_OBJECT FileObject,
72420|     IN BOOLEAN Wait,
72421|     IN PVOID InputBuffer OPTIONAL,
72422|     IN ULONG InputBufferLength,
72423|     OUT PVOID OutputBuffer OPTIONAL,
72424|     IN ULONG OutputBufferLength,
72425|     IN ULONG IoControlCode,
72426|     OUT PIO_STATUS_BLOCK IoStatus,
72427|     IN PDEVICE_OBJECT DeviceObject
72428| )
72429|
72430| /*++
72431|
72432| Routine Description:
72433|
72434| This routine is the fast I/O "pass through" routine
72435| | for device I/O control
72436| operations on a file.
72437|
72438| This function simply invokes the file system's
72439| | cooresponding routine, or
72440| returns FALSE if the file system does not implement
72441| | the function.
72442|
72443| Arguments:
72444| FileObject - Pointer to the file object
72445| | representing the device to be
72446| serviced.

```



```

72444|
72445|    Wait - Indicates whether or not the caller is
       | willing to wait if the
72446|        appropriate locks, etc. cannot be acquired
72447|
72448|    InputBuffer - Optional pointer to a buffer to be
       | passed into the driver.
72449|
72450|    InputBufferLength - Length of the optional
       | InputBuffer, if one was
72451|        specified.
72452|
72453|    OutputBuffer - Optional pointer to a buffer to
       | receive data from the
72454|        driver.
72455|
72456|    OutputBufferLength - Length of the optional
       | OutputBuffer, if one was
72457|        specified.
72458|
72459|    IoControlCode - I/O control code indicating the
       | operation to be performed
72460|        on the device.
72461|
72462|    IoStatus - Pointer to a variable to receive the I/O
       | status of the
72463|        operation.
72464|
72465|    DeviceObject - Pointer to this driver's device
       | object, the device on
72466|        which the operation is to occur.
72467|
72468| Return Value:
72469|
72470|    The function value is TRUE or FALSE based on
       | whether or not fast I/O
72471|        is possible for this file.
72472|
72473| --*/
72474|
72475| {
72476|     PDEVICE_OBJECT deviceObject;
72477|     PFAST_IO_DISPATCH fastIoDispatch;
72478|
72479|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoDevi
       | ceControl")) {
72480|         return FALSE;
72481|     }
72482|

```

```

72483|    deviceObject = ((PFS_FILTER_EXTENSION)
| (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72484|    if ( !deviceObject ) {
72485|        return FALSE;
72486|    }
72487|    fastIoDispatch =
| deviceObject->DriverObject->FastIoDispatch;
72488|
72489|    if ( fastIoDispatch &&
| fastIoDispatch->FastIoDeviceControl ) {
72490|        return(fastIoDispatch->FastIoDeviceControl)(
72491|
| FileObject,
72492|
| Wait,
72493|
| InputBuffer,
72494|
| InputBufferLength,
72495|
| OutputBuffer,
72496|
| OutputBufferLength,
72497|
| IoControlCode,
72498|
| IoStatus,
72499|
| deviceObject
72500|                                     );
72501|    } else {
72502|        return FALSE;
72503|    }
72504| }
72505|
72506| STATIC
72507| VOID
72508| SfFastIoDetachDevice(
72509|         IN PDEVICE_OBJECT SourceDevice,
72510|         IN PDEVICE_OBJECT TargetDevice
72511|         )
72512|
72513| /*++
72514|
72515| Routine Description:
72516|
72517| This routine is invoked on the fast path to detach
| from a device that
72518| is being deleted. This occurs when this driver has
| attached to a file

```

```

72519|    system volume device object, and then, for some
      | reason, the file system
72520|    decides to delete that device (it is being
      | dismantled, it was dismantled
72521|    at some point in the past and its last reference
      | has just gone away, etc.)
72522|
72523| Arguments:
72524|
72525|    SourceDevice - Pointer to this driver's device
      | object, which is attached
72526|    to the file system's volume device object.
72527|
72528|    TargetDevice - Pointer to the file system's volume
      | device object.
72529|
72530| Return Value:
72531|
72532|    The function value is TRUE or FALSE based on
      | whether or not fast I/O
72533|    is possible for this file.
72534|
72535| --*/
72536|
72537| {
72538|    PAGED_CODE();
72539|
72540|    if (
      | (PsmGetObjectType(SourceDevice)!=OBJECT_FS_FILTER) ) {
72541|        Debug(DEBUG_SFILTER,("SFILTER: FastIoDetach,
      | source=%08x is not a FS object, target was
      | %08x\n",SourceDevice,TargetDevice));
72542|        return;
72543|    }
72544|
72545|    Debug(DEBUG_SFILTER,("SFILTER: FastIoDetach,
      | source=%08x,
      | target=%08x\n",SourceDevice,TargetDevice));
72546|
      | ASSERT(PsmGetObjectType(SourceDevice)==OBJECT_FS_FILTER)
      | ;
72547|
72548|    //
72549|    // Simply acquire the database lock for exclusive
      | access, and detach from
72550|    // the file system's volume device object.
72551|    //
72552|
72553|    FsRtlEnterFileSystem();
72554|    ExAcquireResourceExclusive( &FsLock, TRUE );

```

```

72555| IoDetachDevice( TargetDevice );
72556| IoDeleteDevice( SourceDevice );
72557| ExReleaseResource( &FsLock );
72558| FsRtlExitFileSystem();
72559| }
72560|
72561| STATIC
72562| BOOLEAN
72563| SfFastIoQueryNetworkOpenInfo(
72564|             IN PFILE_OBJECT FileObject,
72565|             IN BOOLEAN Wait,
72566|             OUT
| PFILE_NETWORK_OPEN_INFORMATION Buffer,
72567|             OUT PIO_STATUS_BLOCK
| IoStatus,
72568|             IN PDEVICE_OBJECT
| DeviceObject
72569|             )
72570|
72571| /*++
72572|
72573| Routine Description:
72574|
72575| This routine is the fast I/O "pass through" routine
| for querying network
72576| information about a file.
72577|
72578| This function simply invokes the file system's
| corresponding routine, or
72579| returns FALSE if the file system does not implement
| the function.
72580|
72581| Arguments:
72582|
72583| FileObject - Pointer to the file object to be
| queried.
72584|
72585| Wait - Indicates whether or not the caller can
| handle the file system
72586| having to wait and tie up the current thread.
72587|
72588| Buffer - Pointer to a buffer to receive the network
| information about the
72589| file.
72590|
72591| IoStatus - Pointer to a variable to receive the
| final status of the query
72592| operation.
72593|
72594| DeviceObject - Pointer to this driver's device

```

```

    | object, the device on
72595|     which the operation is to occur.
72596|
72597| Return Value:
72598|
72599|     The function value is TRUE or FALSE based on
    | whether or not fast I/O
72600|     is possible for this file.
72601|
72602| --*/
72603|
72604| {
72605|     PDEVICE_OBJECT deviceObject;
72606|     PFAST_IO_DISPATCH fastIoDispatch;
72607|
72608|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoQuer
    | yNetworkOpenInfo")) {
72609|         return FALSE;
72610|     }
72611|
72612|     deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72613|     if ( !deviceObject ) {
72614|         return FALSE;
72615|     }
72616|     fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
72617|
72618|     if ( fastIoDispatch &&
72619|         fastIoDispatch->SizeOfFastIoDispatch >
    | FIELD_OFFSET( FAST_IO_DISPATCH,
    | FastIoQueryNetworkOpenInfo ) &&
72620|         fastIoDispatch->FastIoQueryNetworkOpenInfo ) {
72621|
    | return(fastIoDispatch->FastIoQueryNetworkOpenInfo)(
72622|
    | FileObject,
72623|
    | Wait,
72624|
    | Buffer,
72625|
    | IoStatus,
72626|
    | deviceObject
72627|
    | );
72628|     } else {
72629|         return FALSE;

```

```

72630|    }
72631|
72632| }
72633|
72634| STATIC
72635| BOOLEAN
72636| SfFastIoMdlRead(
72637|     IN PFILE_OBJECT FileObject,
72638|     IN PLARGE_INTEGER FileOffset,
72639|     IN ULONG Length,
72640|     IN ULONG LockKey,
72641|     OUT PMDL *MdlChain,
72642|     OUT PIO_STATUS_BLOCK IoStatus,
72643|     IN PDEVICE_OBJECT DeviceObject
72644| )
72645|
72646| /*++
72647|
72648| Routine Description:
72649|
72650| This routine is the fast I/O "pass through" routine
72651|   | for reading a file
72652|   | using MDLs as buffers.
72653|
72654| This function simply invokes the file system's
72655|   | cooresponding routine, or
72656|   | returns FALSE if the file system does not implement
72657|   | the function.
72658|
72659| Arguments:
72660|
72661| FileObject - Pointer to the file object that is to
72662|   | be read.
72663|
72664| FileOffset - Supplies the offset into the file to
72665|   | begin the read operation.
72666|
72667| Length - Specifies the number of bytes to be read
72668|   | from the file.
72669|
72670| LockKey - The key to be used in byte range lock
72671|   | checks.
72672|
72673| MdlChain - A pointer to a variable to be filled in
72674|   | w/a pointer to the MDL
72675|   | chain built to describe the data read.
72676|
72677| IoStatus - Variable to receive the final status of
72678|   | the read operation.
72679|
72680|

```

```

72671| DeviceObject - Pointer to this driver's device
      | object, the device on
72672|       which the operation is to occur.
72673|
72674| Return Value:
72675|
72676| The function value is TRUE or FALSE based on
      | whether or not fast I/O
72677| is possible for this file.
72678|
72679| --*/
72680|
72681| {
72682|     PDEVICE_OBJECT deviceObject;
72683|     PFAST_IO_DISPATCH fastIoDispatch;
72684|
72685|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoMdlR
      | ead")) {
72686|         return FALSE;
72687|     }
72688|
72689|     deviceObject = ((PFS_FILTER_EXTENSION)
      | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72690|     if ( !deviceObject ) {
72691|         return FALSE;
72692|     }
72693|     fastIoDispatch =
      | deviceObject->DriverObject->FastIoDispatch;
72694|
72695|     if ( fastIoDispatch &&
72696|         fastIoDispatch->SizeOfFastIoDispatch >
      | FIELD_OFFSET( FAST_IO_DISPATCH, MdlRead ) &&
72697|         fastIoDispatch->MdlRead ) {
72698|         return(fastIoDispatch->MdlRead)(
72699|             FileObject,
72700|             FileOffset,
72701|             Length,
72702|             LockKey,
72703|             MdlChain,
72704|             IoStatus,
72705|             deviceObject
72706|         );
72707|     } else {
72708|         return FALSE;
72709|     }
72710|
72711| }
72712|
72713| STATIC

```

```

72714| BOOLEAN
72715| SfFastIoMdlReadComplete(
72716|             IN PFILE_OBJECT FileObject,
72717|             IN PMDL MdlChain,
72718|             IN PDEVICE_OBJECT DeviceObject
72719|             )
72720|
72721| /*++
72722|
72723| Routine Description:
72724|
72725|   This routine is the fast I/O "pass through" routine
72726|   | for completing an
72727|   MDL read operation.
72728|
72729|   This function simply invokes the file system's
72730|   | cooresponding routine, if
72731|   it has one. It should be the case that this
72732|   | routine is invoked only if
72733|   the MdlRead function is supported by the underlying
72734|   | file system, and
72735|   therefore this function will also be supported, but
72736|   | this is not assumed
72737|   by this driver.
72738|
72739| Arguments:
72740|
72741|   FileObject - Pointer to the file object to complete
72742|   | the MDL read upon.
72743|
72744|   MdlChain - Pointer to the MDL chain used to perform
72745|   | the read operation.
72746|
72747|   DeviceObject - Pointer to this driver's device
72748|   | object, the device on
72749|   which the operation is to occur.
72750|
72751| Return Value:
72752|
72753|   The function value is TRUE or FALSE, depending on
72754|   | whether or not it is
72755|   possible to invoke this function on the fast I/O
72756|   | path.
72757|
72758| --*/
72759|
72760| {
72761|   PDEVICE_OBJECT deviceObject;
72762|   PFAST_IO_DISPATCH fastIoDispatch;
72763|

```



```

72754|
72755|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoMdlR
72756|     | eadComplete")) {
72757|         return FALSE;
72758|     }
72759|     deviceObject = ((PFS_FILTER_EXTENSION)
72760|     | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72761|     if ( !deviceObject ) {
72762|         return FALSE;
72763|     }
72764|     fastIoDispatch =
72765|     | deviceObject->DriverObject->FastIoDispatch;
72766|     if ( fastIoDispatch &&
72767|         fastIoDispatch->SizeOfFastIoDispatch >
72768|         | FIELD_OFFSET( FAST_IO_DISPATCH, MdlReadComplete ) &&
72769|         fastIoDispatch->MdlReadComplete ) {
72770|         return(fastIoDispatch->MdlReadComplete)(
72771|         | FileObject,
72772|         | MdlChain,
72773|         | deviceObject
72774|         );
72775|     }
72776|     return FALSE;
72777| }
72778|
72779| STATIC
72780| BOOLEAN
72781| SfFastIoPrepareMdlWrite(
72782|     IN PFILE_OBJECT FileObject,
72783|     IN PLARGE_INTEGER FileOffset,
72784|     IN ULONG Length,
72785|     IN ULONG LockKey,
72786|     OUT PMDL *MdlChain,
72787|     OUT PIO_STATUS_BLOCK IoStatus,
72788|     IN PDEVICE_OBJECT DeviceObject
72789| )
72790| /*++
72791| Routine Description:
72792| This routine is the fast I/O "pass through" routine
72793| | for preparing for an
72794| MDL write operation.

```

```

72795|
72796| This function simply invokes the file system's
    | cooresponding routine, or
72797| returns FALSE if the file system does not implement
    | the function.
72798|
72799| Arguments:
72800|
72801| FileObject - Pointer to the file object that will
    | be written.
72802|
72803| FileOffset - Supplies the offset into the file to
    | begin the write operation.
72804|
72805| Length - Specifies the number of bytes to be write
    | to the file.
72806|
72807| LockKey - The key to be used in byte range lock
    | checks.
72808|
72809| MdlChain - A pointer to a variable to be filled in
    | w/a pointer to the MDL
72810| chain built to describe the data written.
72811|
72812| IoStatus - Variable to receive the final status of
    | the write operation.
72813|
72814| DeviceObject - Pointer to this driver's device
    | object, the device on
72815| which the operation is to occur.
72816|
72817| Return Value:
72818|
72819| The function value is TRUE or FALSE based on
    | whether or not fast I/O
72820| is possible for this file.
72821|
72822| --*/
72823|
72824| {
72825| PDEVICE_OBJECT deviceObject;
72826| PFAST_IO_DISPATCH fastIoDispatch;
72827|
72828| | if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoPrep
    | areMdlWrite")) {
72829| return FALSE;
72830| }
72831|
72832| PFS_FILTER_EXTENSION DevExt =

```

```

    | (PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceObject);
72833|
72834|     if ( (DevExt->Virtual) &&
    | (DevExt->PSMStorageObject) ) {
72835|         PVDISK_EXTENSION
    | VDiskExt=(PVDISK_EXTENSION)GetDeviceExtension(DevExt->PS
    | MStorageObject);
72836|
72837|         if ( VDiskExt->SnapShot ) {
72838|             pPersistentDictionary dictionary =
    | (pPersistentDictionary)VDiskExt->SnapShot->Dictionary;
72839|             ASSERT ( dictionary != NULL );
72840|             if ( dictionary != NULL ) {
72841|                 if ( dictionary->IsReadOnly() ) {
72842|                     if ( FileIsReadOnly(FileObject) ) {
72843|                         // virtual volume is marked
    | readonly, deny this write
72844|                         Debug(DEBUG_SFILTER,("SFILTER:
    | Write: MdlWrite to readonly volume\n"));
72845|                         IoStatus->Information = 0;
72846|                         IoStatus->Status =
    | STATUS_MEDIA_WRITE_PROTECTED;
72847|                         return TRUE;
72848|                     }
72849|                 } else {
72850|                     // not a read-only snapshot. Need
    | to check cache usage...
72851|                     if ( !(gVDiskIOHandling &
    | PSM_VDISK_FLAG_CACHE_FULL_DELETE_SS) ) {
72852|                         if (
    | dictionary->IsCacheWarningThresholdReached() ) {
72853|                             if (
    | FileIsOpenOnSnapShot(FileObject,FALSE) ) {
72854|                                 // Fail the write to
    | the snapshot because too much cache is in use...
72855|                                 NTSTATUS Status =
    | IoStatus->Status = STATUS_DISK_FULL;
72856|                                 IoStatus->Information =
    | 0;
72857|
    | Debug(DEBUG_SFILTER,("SfFastIoPrepareMdlWrite:
    | Reporting STATUS_DISK_FULL; DevExt=%08x,
    | VDiskExt=%08x\n",DevExt,VDiskExt));
72858|                                 return TRUE;
72859|                             }
72860|                         }
72861|                     }
72862|                 }
72863|             }
72864|         }

```

```

72865|    }
72866|
72867|    deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72868|    if ( !deviceObject ) {
72869|        return FALSE;
72870|    }
72871|    fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
72872|
72873|    if ( fastIoDispatch &&
72874|        fastIoDispatch->SizeOfFastIoDispatch >
    | FIELD_OFFSET( FAST_IO_DISPATCH, PrepareMdlWrite ) &&
72875|        fastIoDispatch->PrepareMdlWrite ) {
72876|        return(fastIoDispatch->PrepareMdlWrite)(
72877|
    | FileObject,
72878|
    | FileOffset,
72879|                                     Length,
72880|                                     LockKey,
72881|
    | MdlChain,
72882|
    | IoStatus,
72883|
    | deviceObject
72884|                                     );
72885|    } else {
72886|        return FALSE;
72887|    }
72888|
72889| }
72890|
72891| STATIC
72892| BOOLEAN
72893| SfFastIoMdlWriteComplete(
72894|     IN PFILE_OBJECT FileObject,
72895|     IN PLARGE_INTEGER FileOffset,
72896|     IN PMDL MdlChain,
72897|     IN PDEVICE_OBJECT DeviceObject
72898| )
72899|
72900| /*++
72901|
72902| Routine Description:
72903|
72904| This routine is the fast I/O "pass through" routine
    | for completing an
72905| MDL write operation.

```

```

72906|
72907| This function simply invokes the file system's
    | cooresponding routine, if
72908| it has one. It should be the case that this
    | routine is invoked only if
72909| the PrepareMdlWrite function is supported by the
    | underlying file system,
72910| and therefore this function will also be supported,
    | but this is not
72911| assumed by this driver.
72912|
72913| Arguments:
72914|
72915| FileObject - Pointer to the file object to complete
    | the MDL write upon.
72916|
72917| FileOffset - Supplies the file offset at which the
    | write took place.
72918|
72919| MdlChain - Pointer to the MDL chain used to perform
    | the write operation.
72920|
72921| DeviceObject - Pointer to this driver's device
    | object, the device on
72922| which the operation is to occur.
72923|
72924| Return Value:
72925|
72926| The function value is TRUE or FALSE, depending on
    | whether or not it is
72927| possible to invoke this function on the fast I/O
    | path.
72928|
72929| --*/
72930|
72931| {
72932|     PDEVICE_OBJECT deviceObject;
72933|     PFAST_IO_DISPATCH fastIoDispatch;
72934|
72935|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoMdlW
    | rieComplete")) {
72936|         return FALSE;
72937|     }
72938|
72939|     deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
72940|     if ( !deviceObject ) {
72941|         return FALSE;
72942|     }

```

```

72943| fastIoDispatch =
| deviceObject->DriverObject->FastIoDispatch;
72944|
72945| if ( fastIoDispatch &&
72946|     fastIoDispatch->SizeOfFastIoDispatch >
| FIELD_OFFSET( FAST_IO_DISPATCH, MdlWriteComplete ) &&
72947|     fastIoDispatch->MdlWriteComplete ) {
72948|     return(fastIoDispatch->MdlWriteComplete)(
72949|
| FileObject,
72950|
| FileOffset,
72951|
| MdlChain,
72952|
| deviceObject
72953|         );
72954| }
72955|
72956| return FALSE;
72957| }
72958|
72959| STATIC
72960| BOOLEAN
72961| SfFastIoReadCompressed(
72962|     IN PFILE_OBJECT FileObject,
72963|     IN PLARGE_INTEGER FileOffset,
72964|     IN ULONG Length,
72965|     IN ULONG LockKey,
72966|     OUT PVOID Buffer,
72967|     OUT PMDL *MdlChain,
72968|     OUT PIO_STATUS_BLOCK IoStatus,
72969|     OUT struct _COMPRESSED_DATA_INFO
| *CompressedDataInfo,
72970|     IN ULONG
| CompressedDataInfoLength,
72971|     IN PDEVICE_OBJECT DeviceObject
72972| )
72973|
72974| /*++
72975|
72976| Routine Description:
72977|
72978| This routine is the fast I/O "pass through" routine
| for reading compressed
72979| data from a file.
72980|
72981| This function simply invokes the file system's
| corresponding routine, or
72982| returns FALSE if the file system does not implement

```

| the function.

72983|

72984| Arguments:

72985|

72986|     FileObject - Pointer to the file object that will  
| be read.

72987|

72988|     FileOffset - Supplies the offset into the file to  
| begin the read operation.

72989|

72990|     Length - Specifies the number of bytes to be read  
| from the file.

72991|

72992|     LockKey - The key to be used in byte range lock  
| checks.

72993|

72994|     Buffer - Pointer to a buffer to receive the  
| compressed data read.

72995|

72996|     MdlChain - A pointer to a variable to be filled in  
| w/a pointer to the MDL  
| chain built to describe the data read.

72997|

72998|

72999|     IoStatus - Variable to receive the final status of  
| the read operation.

73000|

73001|     CompressedDataInfo - A buffer to receive the  
| description of the compressed  
| data.

73002|

73003|

73004|     CompressedDataInfoLength - Specifies the size of  
| the buffer described by  
| the CompressedDataInfo parameter.

73005|

73006|

73007|     DeviceObject - Pointer to this driver's device  
| object, the device on  
| which the operation is to occur.

73008|

73009|

73010| Return Value:

73011|

73012|     The function value is TRUE or FALSE based on  
| whether or not fast I/O  
| is possible for this file.

73013|

73014|

73015| --\*/

73016|

73017| {

73018|     PDEVICE\_OBJECT deviceObject;

73019|     PFAST\_IO\_DISPATCH fastIoDispatch;

73020|

```

73021|
    | if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoRead
    | Compressed")) {
73022|         return FALSE;
73023|     }
73024|
73025|     deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73026|     if ( !deviceObject ) {
73027|         return FALSE;
73028|     }
73029|     fastIoDispatch =
        | deviceObject->DriverObject->FastIoDispatch;
73030|
73031|     if ( fastIoDispatch &&
73032|         fastIoDispatch->SizeOfFastIoDispatch >
        | FIELD_OFFSET( FAST_IO_DISPATCH, FastIoReadCompressed )
        | &&
73033|         fastIoDispatch->FastIoReadCompressed ) {
73034|         return(fastIoDispatch->FastIoReadCompressed)(
73035|             | FileObject,
73036|             | FileOffset,
73037|             | Length,
73038|             | LockKey,
73039|             | Buffer,
73040|             | MdlChain,
73041|             | IoStatus,
73042|             | CompressedDataInfo,
73043|             | CompressedDataInfoLength,
73044|             | deviceObject
73045|             );
73046|     } else {
73047|         return FALSE;
73048|     }
73049|
73050| }
73051|
73052| STATIC
73053| BOOLEAN
73054| SfFastIoWriteCompressed(

```



```

73055|          IN PFILE_OBJECT FileObject,
73056|          IN PLARGE_INTEGER FileOffset,
73057|          IN ULONG Length,
73058|          IN ULONG LockKey,
73059|          IN PVOID Buffer,
73060|          OUT PMDL *MdlChain,
73061|          OUT PIO_STATUS_BLOCK IoStatus,
73062|          IN struct _COMPRESSED_DATA_INFO
73063|          | *CompressedDataInfo,
73064|          IN ULONG
73065|          | CompressedDataInfoLength,
73066|          IN PDEVICE_OBJECT DeviceObject
73067|          )
73068|
73069| Routine Description:
73070|
73071| This routine is the fast I/O "pass through" routine
73072| | for writing compressed
73073| data to a file.
73074| This function simply invokes the file system's
73075| | cooresponding routine, or
73076| returns FALSE if the file system does not implement
73077| | the function.
73078|
73079| Arguments:
73080|
73081| FileObject - Pointer to the file object that will
73082| | be written.
73083|
73084| FileOffset - Supplies the offset into the file to
73085| | begin the write operation.
73086|
73087| Length - Specifies the number of bytes to be write
73088| | to the file.
73089|
73090| LockKey - The key to be used in byte range lock
73091| | checks.
73092|
73093| Buffer - Pointer to the buffer containing the data
73094| | to be written.
73095|
73096| MdlChain - A pointer to a variable to be filled in
73097| | w/a pointer to the MDL
73098| chain built to describe the data written.
73099|
73100| IoStatus - Variable to receive the final status of
73101| | the write operation.

```



```

    | Write: WriteCompressed to readonly volume\n"));
73131|         IoStatus->Information = 0;
73132|         IoStatus->Status =
    | STATUS_MEDIA_WRITE_PROTECTED;
73133|         return TRUE;
73134|     }
73135|     } else {
73136|         // not a read-only snapshot. Need
    | to check cache usage...
73137|         if ( !(gVdiskIOHandling &
    | PSM_VDISK_FLAG_CACHE_FULL_DELETE_SS) ) {
73138|             if (
    | dictionary->IsCacheWarningThresholdReached() ) {
73139|                 if (
    | FileIsOpenOnSnapShot(FileObject,FALSE) ) {
73140|                     // Fail the write to
    | the snapshot because too much cache is in use...
73141|                     NTSTATUS Status =
    | IoStatus->Status = STATUS_DISK_FULL;
73142|                     IoStatus->Information =
    | 0;
73143|                     Debug(DEBUG_SFILTER,("SfFastIoWriteCompressed:
    | Reporting STATUS_DISK_FULL; DevExt=%08x,
    | VDiskExt=%08x\n",DevExt, VDiskExt));
73144|                     return TRUE;
73145|                 }
73146|             }
73147|         }
73148|     }
73149| }
73150| }
73151| }
73152|
73153| deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73154| if ( !deviceObject ) {
73155|     return FALSE;
73156| }
73157| fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
73158|
73159| if ( fastIoDispatch &&
73160|     fastIoDispatch->SizeOfFastIoDispatch >
    | FIELD_OFFSET( FAST_IO_DISPATCH, FastIoWriteCompressed )
    | &&
73161|     fastIoDispatch->FastIoWriteCompressed ) {
73162|     return(fastIoDispatch->FastIoWriteCompressed)(
73163|         FileObject,

```

```

73164|
73165|     | FileOffset,
73166|     | Length,
73167|     | LockKey,
73168|     | Buffer,
73169|     | MdlChain,
73170|     | IoStatus,
73171|     | CompressedDataInfo,
73172|     | CompressedDataInfoLength,
73173|     | deviceObject
73174|         );
73175|     } else {
73176|         return FALSE;
73177|     }
73178| }
73179|
73180| STATIC
73181| BOOLEAN
73182| SfFastIoMdlReadCompleteCompressed(
73183|     IN PFILE_OBJECT
73184|     | FileObject,
73185|     IN PMDL MdlChain,
73186|     IN PDEVICE_OBJECT
73187|     | DeviceObject
73188| )
73189|
73190| /*++
73191| Routine Description:
73192| This routine is the fast I/O "pass through" routine
73193| for completing an
73194| MDL read compressed operation.
73195| This function simply invokes the file system's
73196| corresponding routine, if
73197| it has one. It should be the case that this
73198| routine is invoked only if
73199| the read compressed function is supported by the
73200| underlying file system,
73201| and therefore this function will also be supported,

```

```

    | but this is not assumed
73199|   by this driver.
73200|
73201| Arguments:
73202|
73203|   FileObject - Pointer to the file object to complete
    | the compressed read
73204|   upon.
73205|
73206|   MdlChain - Pointer to the MDL chain used to perform
    | the read operation.
73207|
73208|   DeviceObject - Pointer to this driver's device
    | object, the device on
73209|   which the operation is to occur.
73210|
73211| Return Value:
73212|
73213|   The function value is TRUE or FALSE, depending on
    | whether or not it is
73214|   possible to invoke this function on the fast I/O
    | path.
73215|
73216| --*/
73217|
73218| {
73219|   PDEVICE_OBJECT deviceObject;
73220|   PFAST_IO_DISPATCH fastIoDispatch;
73221|
73222|   | if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoMdlR
    | eadCompleteCompressed")) {
73223|       return FALSE;
73224|   }
73225|
73226|   deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73227|   if ( !deviceObject ) {
73228|       return FALSE;
73229|   }
73230|   fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
73231|
73232|   if ( fastIoDispatch &&
73233|       fastIoDispatch->SizeOfFastIoDispatch >
    | FIELD_OFFSET( FAST_IO_DISPATCH,
    | MdlReadCompleteCompressed ) &&
73234|       fastIoDispatch->MdlReadCompleteCompressed ) {
73235|       | return(fastIoDispatch->MdlReadCompleteCompressed)(

```

```

73236|
73237|     | FileObject,
73238|     | MdlChain,
73239|     | deviceObject
73240|     | );
73241| }
73242|     return FALSE;
73243| }
73244|
73245| STATIC
73246| BOOLEAN
73247| SfFastIoMdlWriteCompleteCompressed(
73248|     IN PFILE_OBJECT
73249|     | FileObject,
73250|     IN PLARGE_INTEGER
73251|     | FileOffset,
73252|     IN PMDL MdlChain,
73253|     IN PDEVICE_OBJECT
73254|     | DeviceObject
73255|     )
73256| /*++
73257| Routine Description:
73258| This routine is the fast I/O "pass through" routine
73259| for completing a
73260| write compressed operation.
73261| This function simply invokes the file system's
73262| corresponding routine, if
73263| it has one. It should be the case that this
73264| routine is invoked only if
73265| the write compressed function is supported by the
73266| underlying file system,
73267| and therefore this function will also be supported,
73268| but this is not assumed
73269| by this driver.
73270| Arguments:
73271| FileObject - Pointer to the file object to complete
73272| the compressed write
73273| upon.
73274| FileOffset - Supplies the file offset at which the

```

```

    | file write operation
73273|     began.
73274|
73275|     MdlChain - Pointer to the MDL chain used to perform
    | the write operation.
73276|
73277|     DeviceObject - Pointer to this driver's device
    | object, the device on
73278|     which the operation is to occur.
73279|
73280| Return Value:
73281|
73282|     The function value is TRUE or FALSE, depending on
    | whether or not it is
73283|     possible to invoke this function on the fast I/O
    | path.
73284|
73285| --*/
73286|
73287| {
73288|     PDEVICE_OBJECT deviceObject;
73289|     PFAST_IO_DISPATCH fastIoDispatch;
73290|
73291|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoMdlW
    | riteCompleteCompressed")) {
73292|         return FALSE;
73293|     }
73294|
73295|     deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73296|     if ( !deviceObject ) {
73297|         return FALSE;
73298|     }
73299|     fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
73300|
73301|     if ( fastIoDispatch &&
73302|         fastIoDispatch->SizeOfFastIoDispatch >
    | FIELD_OFFSET( FAST_IO_DISPATCH,
    | MdlWriteCompleteCompressed ) &&
73303|         fastIoDispatch->MdlWriteCompleteCompressed ) {
73304|         | return(fastIoDispatch->MdlWriteCompleteCompressed)(
73305|         | FileObject,
73306|         | FileOffset,
73307|         | MdlChain,

```

```

73308|
73309|     | deviceObject
73310|     | );
73311|     }
73312|     return FALSE;
73313| }
73314|
73315| STATIC
73316| BOOLEAN
73317| SfFastIoQueryOpen(
73318|     IN PIRP Irp,
73319|     OUT PFILE_NETWORK_OPEN_INFORMATION
73320|     | NetworkInformation,
73321|     IN PDEVICE_OBJECT DeviceObject
73322|     )
73323| /*++
73324|
73325| Routine Description:
73326|
73327| This routine is the fast I/O "pass through" routine
73328| | for opening a file
73329| | and returning network information it.
73330| This function simply invokes the file system's
73331| | cooresponding routine, or
73332| | returns FALSE if the file system does not implement
73333| | the function.
73334|
73335| Arguments:
73336| Irp - Pointer to a create IRP that represents this
73337| | open operation. It is
73338| | to be used by the file system for common
73339| | open/create code, but not
73340| | actually completed.
73341|
73342| NetworkInformation - A buffer to receive the
73343| | information required by the
73344| | network about the file being opened.
73345|
73346| DeviceObject - Pinter to this driver's device
73347| | object, the device on
73348| | which the operation is to occur.
73349|
73350| Return Value:
73351|
73352| The function value is TRUE or FALSE based on

```



```

    | whether or not fast I/O
73348|    is possible for this file.
73349|
73350|    **/
73351|
73352| {
73353|     PDEVICE_OBJECT deviceObject;
73354|     PFAST_IO_DISPATCH fastIoDispatch;
73355|     BOOLEAN result;
73356|
73357|     PAGED_CODE();
73358|
73359|     //Debug(DEBUG_SFILTER,("SFILTER:
    | FastIoQueryOpen\n"));
73360|
73361|     if (
        | PsmGetObjectTypes(DeviceObject)!=OBJECT_FS_FILTER ) {
73362|         Debug(DEBUG_SFILTER,("SFILTER: FastIoQueryOpen:
    | Not fs object\n"));
73363|         return FALSE;
73364|     }
73365|
73366|     | ASSERT(PsmGetObjectTypes(DeviceObject)==OBJECT_FS_FILTER)
    | ;
73367|     deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73368|     if ( !deviceObject ) {
73369|         return FALSE;
73370|     }
73371|     fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
73372|
73373|     if ( fastIoDispatch &&
73374|         fastIoDispatch->SizeOfFastIoDispatch >
    | FIELD_OFFSET( FAST_IO_DISPATCH, FastIoQueryOpen ) &&
73375|         fastIoDispatch->FastIoQueryOpen ) {
73376|         PIO_STACK_LOCATION irpSp =
    | IoGetCurrentIrpStackLocation( Irp );
73377|
73378|         irpSp->DeviceObject = deviceObject;
73379|
73380|         result = (fastIoDispatch->FastIoQueryOpen)(
73381|             Irp,
73382|
    | NetworkInformation,
73383|
    | deviceObject
73384|             );
73385|         if ( !result ) {

```

```

73386|         irpSp->DeviceObject = DeviceObject;
73387|     }
73388|     return result;
73389| } else {
73390|     return FALSE;
73391| }
73392| }
73393|
73394|
73395|
73396| STATIC
73397| NTSTATUS
73398| SfMountCompletion(
73399|     IN PDEVICE_OBJECT DeviceObject,
73400|     IN PIRP Irp,
73401|     IN PVOID Context
73402| )
73403|
73404| /*++
73405|
73406| Routine Description:
73407|
73408| This routine is invoked for the completion of a
73409| | mount request. If the
73410| | mount was successful, then this file system
73411| | attaches its device object to
73412| | the file system's volume device object. Otherwise,
73413| | the interim device
73414| | object is deleted.
73415|
73416| Arguments:
73417|
73418| DeviceObject - Pointer to this driver's device
73419| | object.
73420|
73421| Irp - Pointer to the IRP that was just completed.
73422|
73423| Context - Pointer to the saved device object to
73424| | attach.
73425|
73426| Return Value:
73427|
73428| The return value is always STATUS_SUCCESS.
73429|
73430| --*/
73431| {
73432|     NTSTATUS Status = STATUS_SUCCESS;
73433|     PDEVICE_OBJECT fsfDeviceObject = (PDEVICE_OBJECT)
73434| | Context;

```

```

73430|   PFS_FILTER_EXTENSION deviceExtension =
      | (PFS_FILTER_EXTENSION)GetDeviceExtension(fsfDeviceObject
      | );
73431|   PIO_STACK_LOCATION irpSp =
      | IoGetCurrentIrpStackLocation( Irp );
73432|   PDEVICE_OBJECT deviceObject = 0;
73433|   PVPB vpb = 0;
73434|
73435|   | ASSERT(PsmGetObjectTypes(DeviceObject)==OBJECT_FS_FILTER)
      | ;
73436|   #if DO_ALL_SFILTER
73437|       Debug(DEBUG_SFILTER,("SFILTER: MountCompletion
      | %08x, %08x,
      | %08x-%08x\n",DeviceObject,Irp,Irp->IoStatus.Status,Irp->
      | IoStatus.Information));
73438|   #endif /*DO_ALL_SFILTER*/
73439|
73440|   FsRtlEnterFileSystem();
73441|   ExAcquireResourceExclusive( &FsLock, TRUE );
73442|
73443|   //
73444|   // Determine whether or not the request was
      | successful and act accordingly.
73445|   //
73446|
73447|   if ( NT_SUCCESS( Irp->IoStatus.Status ) ) {
73448|
73449|       //
73450|       // Note that the VPB must be picked up from the
      | target device object
73451|       // in case the file system did a remount of a
      | previous volume, in
73452|       // which case it has replaced the VPB passed in
      | as the target with
73453|       // a previously mounted VPB. Note also that in
      | the mount dispatch
73454|       // routine, this driver *replaced* the
      | DeviceObject pointer with a
73455|       // pointer to the real device, not the device
      | that the file system
73456|       // was supposed to talk to, since this driver
      | does not care.
73457|       //
73458|       #if DO_ALL_SFILTER
73459|           Debug(DEBUG_SFILTER,("SFILTER: 2 MyDo=%08x,
      | fsfdo=%08x, mvpb=%08x, mdo=%08x, vrd=%08x,
      | vdo=%08x,vfl=%08x\n",
73460|                               DeviceObject,
73461|                               fsfDeviceObject,

```

```

73462|
| irpSp->Parameters.MountVolume.Vpb,
73463|
| irpSp->Parameters.MountVolume.DeviceObject,
73464|
| irpSp->Parameters.MountVolume.Vpb->RealDevice,
73465|
| irpSp->Parameters.MountVolume.Vpb->DeviceObject,
73466|
| irpSp->Parameters.MountVolume.Vpb->Flags
73467|         ));
73468|     #endif /*DO_ALL_SFILTER*/
73469|
73470|     vpb =
| irpSp->Parameters.MountVolume.DeviceObject->Vpb;
73471|
73472|     PDEVICE_OBJECT PSMStorageObject =
| GetPSMStorageFilterObject(vpb);
73473|     if(!PSMStorageObject) {
73474|         // this can happen when the volume being
| mounted is not a volume we filter
73475|         // for example, A: or B:
73476|         Debug(DEBUG_SFILTER,("SFILTER: 2 Error!
| Unable to find lower object, not attaching to
| volume!\n"));
73477|         goto ExitMount;
73478|     }
73479|
73480|     deviceObject = IoAttachDeviceToDeviceStack(
| (PDEVICE_OBJECT) Context, vpb->DeviceObject );
73481|
73482|     if ( deviceObject == NULL ) {
73483|         Debug(DEBUG_SFILTER,("SFILTER: 2 Attached
| failed\n"));
73484| ExitMount:
73485|         IoDeleteDevice( (PDEVICE_OBJECT) Context );
73486|         ExReleaseResource( &FsLock );
73487|         FsRtlExitFileSystem();
73488|
73489|         //
73490|         // If pending was returned, then propagate
| it to the caller.
73491|         //
73492|
73493|         if ( Irp->PendingReturned ) {
73494|             IoMarkIrpPending( Irp );
73495|         }
73496|
73497|         return STATUS_SUCCESS;
73498|     }

```

```

73499|
73500|     deviceExtension->TargetDeviceObject =
| deviceObject;
73501|     deviceExtension->Attached = TRUE;
73502|     deviceExtension->PSMStorageObject =
| PSMStorageObject;
73503|     deviceExtension->Virtual = FALSE;
73504|     deviceExtension->FileSystem = FALSE;
73505|
73506|
73507|     if ( deviceExtension->PSMStorageObject ) {
73508|         if (
| PsmGetObjectTypes(deviceExtension->PSMStorageObject) ==
| OBJECT_FILTEREDDISK ) {
73509|             PFILTERED_EXTENSION d =
| (PFILTERED_EXTENSION)GetDeviceExtension(deviceExtension-
| >PSMStorageObject);
73510|             Debug(DEBUG_SFILTER,("SFILTER: 2 Lower
| object is filtered disk %08x, volume is completely
| mounted %08x\n",PSMStorageObject,d->IsMounted));
73511|             d->IsMounted = TRUE;
73512|             Debug(DEBUG_SFILTER,("SFILTER: Just set
| IsMounted to TRUE for DevExt=%08x,
| DevObj=%08x\n",d,deviceExtension->PSMStorageObject));
73513|         } else
73514|         if (
| PsmGetObjectTypes(deviceExtension->PSMStorageObject) ==
| OBJECT_VIRTUALDISK ) {
73515|             Debug(DEBUG_SFILTER,("SFILTER: 2 Lower
| object is virtual disk %08x\n",PSMStorageObject));
73516|             deviceExtension->Virtual = TRUE;
73517|         } else {
73518| #ifdef DEBUG
73519|             Debug(DEBUG_SFILTER,("SFILTER: 2 Error!
| Unknown lower object!!!!!!!!!!!!!!!!\n"));
73520|             DbgBreakPoint();
73521| #endif
73522|             deviceExtension->PSMStorageObject =
| NULL;
73523|         }
73524|     } else {
73525|         // this can happen when the volume being
| mounted is not a volume we filter
73526|         // for example, A: or B:
73527|         Debug(DEBUG_SFILTER,("SFILTER: 2 Error!
| Unable to find lower object, detaching from
| volume!\n"));
73528|         IoDetachDevice(deviceObject);
73529|         goto ExitMount;
73530|     }

```

```

73531|
73532|     if ( deviceObject->Flags & DO_BUFFERED_IO ) {
73533|         fsfDeviceObject->Flags |= DO_BUFFERED_IO;
73534|     }
73535|
73536|     if ( deviceObject->Flags & DO_DIRECT_IO ) {
73537|         fsfDeviceObject->Flags |= DO_DIRECT_IO;
73538|     }
73539|
73540|     ((PDEVICE_OBJECT) Context)->Flags &=
        | ~DO_DEVICE_INITIALIZING;
73541|     Debug(DEBUG_SFILTER,("SFILTER: 2 Successfully
        | attached to %08x,
        | Mysco=%08x\n",deviceObject,deviceExtension->PSMStorageOb
        | ject));
73542|
73543|         // if we are past system boot, go ahead
        | and map in drives now
73544|         // otherwise, it will occur when the system is
        | ready (basically after chkdsk runs)
73545|         ASSERT(PSMStorageObject);
73546|
73547|         if( (!deviceExtension->Virtual) &&
73548|             (PersistentDictionary::GetSystemReady())
73549|             ) {
73550|             PFILTERED_EXTENSION
        | DevExt=(PFILTERED_EXTENSION)GetDeviceExtension(PSMStorag
        | eObject);
73551|             if(!DevExt->IsPhysical) {
73552|                 HANDLE TempHandle;
73553|
73554|                 Debug(DEBUG_SFILTER,("SFILTER: 2
        | Starting part2 of rebuild\n"));
73555|                 pmStartThread(
73556|
        | (PKSTART_ROUTINE)PersistentDictionary::Part2OfRebuildFor
        | Volume, // IN PKSTART_ROUTINE StartRoutine,
73557|                 (PVOID)PSMStorageObject,
        | // IN PVOID StartContext
73558|                 &TempHandle //
        | OUT PHANDLE ThreadHandle,
73559|                 );
73560|
73561|                 ZwClose(TempHandle);
73562|             } else {
73563|                 Debug(DEBUG_SFILTER,("SFILTER: 2 Device
        | is physical\n"));
73564|             }
73565|         } else {
73566|             if(PersistentDictionary::GetSystemReady())

```

```

| {
73567|         Debug(DEBUG_SFILTER,("SFILTER: 2
| virtual\n"));
73568|     } else {
73569|         Debug(DEBUG_SFILTER,("SFILTER: 2 not
| ready\n"));
73570|     }
73571| }
73572|
73573| } else {
73574|     #if DO_ALL_SFILTER
73575|         Debug(DEBUG_SFILTER,("SFILTER: Mount failed
| %08x\n",Irp->IoStatus.Status));
73576|     #endif /*DO_ALL_SFILTER*/
73577|
73578|     //
73579|     // The mount request failed. Simply delete the
| device object that was
73580|     // created in case this request succeeded.
73581|     //
73582|
73583|     IoDeleteDevice( (PDEVICE_OBJECT) Context );
73584| }
73585|
73586|     ExReleaseResource( &FsLock );
73587|     FsRtlExitFileSystem();
73588|
73589|
73590|     //
73591|     // If pending was returned, then propagate it to
| the caller.
73592|     //
73593|
73594|     if ( Irp->PendingReturned ) {
73595|         IoMarkIrpPending( Irp );
73596|     }
73597|
73598|     return Status;
73599| }
73600|
73601| STATIC
73602| NTSTATUS
73603| SfLoadFsCompletion(
73604|     IN PDEVICE_OBJECT DeviceObject,
73605|     IN PIRP Irp,
73606|     IN PVOID Context
73607| )
73608|
73609| /*++
73610|

```

```

73611| Routine Description:
73612|
73613|   This routine is invoked upon completion of an FSCTL
       | function to load a
73614|   file system driver (as the result of a mini-file
       | system recognizer seeing
73615|   that an on-disk structure belonged to it). A
       | device object has been
73616|   created by this driver (DeviceObject) so that it
       | can be attached to the
73617|   newly loaded file system. If the load failed, then
       | the device must be
73618|   deleted, but cannot be done here, so it is put on a
       | work queue to be dealt
73619|   with later.
73620|
73621| Arguments:
73622|
73623|   DeviceObject - Pointer to this driver's device
       | object.
73624|
73625|   Irp - Pointer to the I/O Request Packet
       | representing the file system
73626|   driver load request.
73627|
73628|   Context - Context parameter for this driver,
       | unused.
73629|
73630| Return Value:
73631|
73632|   The function value for this routine is always
       | success.
73633|
73634| --*/
73635|
73636| {
73637|   PFS_FILTER_EXTENSION deviceExtension =
       | (PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceObject);
73638|
73639|   //
73640|   // Begin by determining whether or not the load
       | file system request was
73641|   // completed successfully.
73642|   //
73643|
73644|   | ASSERT(PsmGetObjectype(DeviceObject)==OBJECT_FS_FILTER)
       | ;
73645|   if ( !NT_SUCCESS( Irp->IoStatus.Status ) ) {
73646|       Debug(DEBUG_SFILTER,("SFILTER: Load failed

```



```

    | %08x\n", Irp->IoStatus.Status));
73647|
73648|    //
73649|    // The load was not successful. Simply
    | reattach to the recognizer
73650|    // driver in case it ever figures out how to
    | get the driver loaded
73651|    // on a subsequent call.
73652|    //
73653|
73654|    IoAttachDeviceToDeviceStack( DeviceObject,
    | deviceExtension->TargetDeviceObject );
73655|    deviceExtension->Attached = TRUE;
73656| } else {
73657|
73658|    Debug(DEBUG_SFILT,("SFILT: Load
    | successful\n"));
73659|    //
73660|    // The load was successful. However, in order
    | to ensure that these
73661|    // drivers do not go away, the I/O system has
    | artificially bumped the
73662|    // reference count on all parties involved in
    | this maneuver. Therefore,
73663|    // simply remember to delete this device object
    | at some point in the
73664|    // future when its reference count is zero.
73665|    //
73666|
73667|    FsRtlEnterFileSystem();
73668|    ExAcquireResourceExclusive( &FsLock, TRUE );
73669|    InsertTailList(
73670|        &FsDeviceQueue,
73671|        &DeviceObject->Queue.ListEntry
73672|    );
73673|    ExReleaseResource( &FsLock );
73674|    FsRtlExitFileSystem();
73675| }
73676|
73677|    //
73678|    // If pending was returned, then propagate it to
    | the caller.
73679|    //
73680|
73681|    if ( Irp->PendingReturned ) {
73682|        IoMarkIrpPending( Irp );
73683|    }
73684|
73685|    return STATUS_SUCCESS;
73686| }

```

```

73687|
73688|
73689| STATIC
73690| NTSTATUS
73691| SfFastIoAcquireForModWrite(
73692|             IN PFILE_OBJECT
    | FileObject,
73693|             IN PLARGE_INTEGER
    | EndingOffset,
73694|             OUT PERESOURCE
    | *ResourceToRelease,
73695|             IN PDEVICE_OBJECT
    | DeviceObject
73696|             )
73697| /*++
73698|
73699| Routine Description:
73700|
73701| This routine is the fast I/O "pass through" routine
    | for acquiring the
73702| file resource prior to attempting a modified write
    | operation.
73703|
73704| This function simply invokes the next driver's
    | cooresponding routine, or
73705| returns FALSE if the next driver does not implement
    | the function.
73706|
73707| Arguments:
73708|
73709| FileObject - Pointer to the file object whose
    | resource is to be acquired.
73710|
73711| EndingOffset - The offset to the last byte being
    | written plus one.
73712|
73713| ResourceToRelease - Pointer to a variable to return
    | the resource to
73714| release. Not defined if an error is returned.
73715|
73716| DeviceObject - Pointer to this driver's device
    | object, the device on
73717| which the operation is to occur.
73718|
73719| Return Value:
73720|
73721| The function value is either success or failure
    | based on whether or not
73722| fast I/O is possible for this file.
73723|

```

```

73724| --*/
73725| {
73726|     PDEVICE_OBJECT deviceObject;
73727|     PFAST_IO_DISPATCH fastIoDispatch;
73728|
73729|     if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoAcqu
| ireForModWrite")) {
73730|         return FALSE;
73731|     }
73732|
73733|     deviceObject = ((PFS_FILTER_EXTENSION)
| (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73734|     if ( !deviceObject ) {
73735|         return FALSE;
73736|     }
73737|     fastIoDispatch =
| deviceObject->DriverObject->FastIoDispatch;
73738|
73739|     if ( fastIoDispatch &&
73740|         fastIoDispatch->SizeOfFastIoDispatch >
| FIELD_OFFSET( FAST_IO_DISPATCH, AcquireForModWrite) &&
73741|         fastIoDispatch->AcquireForModWrite ) {
73742|         return(fastIoDispatch->AcquireForModWrite)(
73743|             | FileObject,
73744|             | EndingOffset,
73745|             | ResourceToRelease,
73746|             | deviceObject
73747|             );
73748|     }
73749|
73750|     return FALSE;
73751| }
73752|
73753| STATIC
73754| NTSTATUS
73755| SfFastIoReleaseForModWrite(
73756|     IN PFILE_OBJECT FileObject,
73757|     IN PERESOURCE
| ResourceToRelease,
73758|     IN PDEVICE_OBJECT
| DeviceObject
73759| )
73760| /*++
73761|
73762| Routine Description:

```

```

73763|
73764| This routine is the fast I/O "pass through" routine
    | for releasing the
73765| resource previously acquired for performing a
    | modified write operation
73766| to a file.
73767|
73768| This function simply invokes the next driver's
    | corresponding routine, or
73769| returns FALSE if the next driver does not implement
    | the function.
73770|
73771| Arguments:
73772|
73773| FileObject - Pointer to the file object whose
    | resource is to be released.
73774|
73775| ResourceToRelease - Specifies the modified writer
    | resource for the file
73776| that is to be released.
73777|
73778| DeviceObject - Pointer to this driver's device
    | object, the device on
73779| which the operation is to occur.
73780|
73781| Return Value:
73782|
73783| The function value is either success or failure
    | based on whether or not
73784| fast I/O is possible for this file.
73785|
73786| --*/
73787| {
73788| PDEVICE_OBJECT deviceObject;
73789| PFAST_IO_DISPATCH fastIoDispatch;
73790|
73791| | if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoRele
    | aseForModWrite")) {
73792| return FALSE;
73793| }
73794|
73795| deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73796| if ( !deviceObject ) {
73797| return FALSE;
73798| }
73799| fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
73800|

```

```

73801|    if ( fastIoDispatch &&
73802|        fastIoDispatch->SizeOfFastIoDispatch >
73803|        | FIELD_OFFSET( FAST_IO_DISPATCH, ReleaseForModWrite) &&
73804|        fastIoDispatch->ReleaseForModWrite ) {
73805|        return(fastIoDispatch->ReleaseForModWrite)(
73806|            | FileObject,
73807|            | ResourceToRelease,
73808|            | deviceObject
73809|            );
73810|    }
73811|    return FALSE;
73812| }
73813|
73814| STATIC
73815| NTSTATUS
73816| SfFastIoAcquireForCcFlush(
73817|     IN PFILE_OBJECT FileObject,
73818|     IN PDEVICE_OBJECT DeviceObject
73819| )
73820| /*++
73821|
73822| Routine Description:
73823|
73824| This routine is the fast I/O "pass through" routine
73825| for acquiring the
73826| appropriate file system resource prior to a call to
73827| CcFlush.
73828|
73829| This function simply invokes the next driver's
73830| cooresponding routine, or
73831| returns FALSE if the next driver does not implement
73832| the function.
73833|
73834| Arguments:
73835|
73836| FileObject - Pointer to the file object whose
73837| resource is to be acquired.
73838|
73839| DeviceObject - Pointer to this driver's device
73840| object, the device on
73841| which the operation is to occur.
73842|
73843| Return Value:
73844|
73845| The function value is either success or failure
73846| based on whether or not

```

```

73840|    fast I/O is possible for this file.
73841|
73842|  --*/
73843|  {
73844|    PDEVICE_OBJECT deviceObject;
73845|    PFAST_IO_DISPATCH fastIoDispatch;
73846|
73847|    if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoAcqu
    ireForCcFlush")) {
73848|        return FALSE;
73849|    }
73850|
73851|    deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73852|    if ( !deviceObject ) {
73853|        return FALSE;
73854|    }
73855|    fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
73856|
73857|    if ( fastIoDispatch &&
73858|        fastIoDispatch->SizeOfFastIoDispatch >
    | FIELD_OFFSET( FAST_IO_DISPATCH, AcquireForCcFlush) &&
73859|        fastIoDispatch->AcquireForCcFlush ) {
73860|        return(fastIoDispatch->AcquireForCcFlush)(
73861|
    | FileObject,
73862|
    | deviceObject
73863|
    | );
73864|    }
73865|
73866|    return FALSE;
73867| }
73868|
73869| STATIC
73870| NTSTATUS
73871| SfFastIoReleaseForCcFlush(
73872|
    IN PFILE_OBJECT FileObject,
73873|
    IN PDEVICE_OBJECT DeviceObject
73874|
    )
73875| /*++
73876|
73877| Routine Description:
73878|
73879| This routine is the fast I/O "pass through" routine
    | for releasing the
73880| appropriate file system resource previously
    | acquired for a CcFlush.

```

```

73881|
73882| This function simply invokes the next driver's
    | cooresponding routine, or
73883| returns FALSE if the next driver does not implement
    | the function.
73884|
73885| Arguments:
73886|
73887| FileObject - Pointer to the file object whose
    | resource is to be released.
73888|
73889| DeviceObject - Pointer to this driver's device
    | object, the device on
73890| which the operation is to occur.
73891|
73892| Return Value:
73893|
73894| The function value is either success or failure
    | based on whether or not
73895| fast I/O is possible for this file.
73896|
73897| --*/
73898| {
73899| PDEVICE_OBJECT deviceObject;
73900| PFAST_IO_DISPATCH fastIoDispatch;
73901|
73902| if(FastIoCommonStuff(DeviceObject,FileObject,"FastIoRele
    | aseForCcFlush")) {
73903| return FALSE;
73904| }
73905|
73906| deviceObject = ((PFS_FILTER_EXTENSION)
    | (GetDeviceExtension(DeviceObject)))->TargetDeviceObject;
73907| if ( !deviceObject ) {
73908| return FALSE;
73909| }
73910| fastIoDispatch =
    | deviceObject->DriverObject->FastIoDispatch;
73911|
73912| if ( fastIoDispatch &&
73913| fastIoDispatch->SizeOfFastIoDispatch >
    | FIELD_OFFSET( FAST_IO_DISPATCH, ReleaseForCcFlush) &&
73914| fastIoDispatch->ReleaseForCcFlush ) {
73915| return(fastIoDispatch->ReleaseForCcFlush)(
73916| FileObject,
73917| deviceObject
73918| );

```

```

73919|    }
73920|
73921|    return FALSE;
73922| }
73923|
73924| STATIC
73925| NTSTATUS
73926| QueryInformationCompletionRoutine(
73927|     IN PDEVICE_OBJECT
73928|     | DeviceObject,
73929|     IN PIRP Irp,
73930|     IN PVOID Context
73931|     )
73932| {
73933|     PIO_STACK_LOCATION IrpSp =
73934|         | IoGetCurrentIrpStackLocation( Irp );
73935|     if(!NT_SUCCESS(Irp->IoStatus.Status)) {
73936|         Debug(DEBUG_SFILTER,("SFILTER:
73937|         | QueryInformationCompletion: Error
73938|         | %08x\n",Irp->IoStatus.Status));
73939|         return STATUS_SUCCESS;
73940|     }
73941|     | switch(IrpSp->Parameters.SetFile.FileInformationClass)
73942|     | {
73943|     case FileDirectoryInformation : {
73944|         PFILE_DIRECTORY_INFORMATION
73945|         | Info=(PFILE_DIRECTORY_INFORMATION)Irp->AssociatedIrp.Sys
73946|         | temBuffer;
73947|         Debug(DEBUG_SFILTER,("SFILTER: Get
73948|         | FileDirectoryInformation for PSM file
73949|         | %08x\n",IrpSp->FileObject));
73950|         Debug(DEBUG_SFILTER,("SFILTER:  Cr=%I64x,
73951|         | A=%I64x, W=%I64x, Ch=%I64x\n"
73952|         | "SFILTER:  NEO=%08x, FI=%08x,
73953|         | eof=%I64x, alloc=%I64x, attr=%08x\n"
73954|         | "SFILTER:  fnl=%08x, '%-*.*S'\n",
73955|         Info->CreationTime,
73956|         Info->LastAccessTime,
73957|         Info->LastWriteTime,
73958|         Info->ChangeTime,
73959|         Info->NextEntryOffset,
73960|         Info->FileIndex,
73961|         Info->EndOfFile,
73962|         Info->AllocationSize,
73963|         Info->FileAttributes,
73964|         Info->FileNameLength /

```



```

    | sizeof(WCHAR),Info->FileNameLength /
    | sizeof(WCHAR),Info->FileNameLength / sizeof(WCHAR),
73957|         Info->FileName
73958|     ));
73959|
73960|
73961|         break;
73962|     }
73963|     case FileFullDirectoryInformation : {
73964|         PFILE_FULL_DIR_INFORMATION
    | Info=(PFILE_FULL_DIR_INFORMATION)Irp->AssociatedIrp.Syst
    | emBuffer;
73965|         Debug(DEBUG_SFILTER,("SFILTER: Get
    | FileFullDirectoryInformation for PSM file
    | %08x\n",IrpSp->FileObject));
73966|         Debug(DEBUG_SFILTER,("SFILTER:  Cr=%l64x,
    | A=%l64x, W=%l64x, Ch=%l64x\n"
73967|         "SFILTER:  NEO=%08x, FI=%08x,
    | eof=%l64x, alloc=%l64x, attr=%08x\n"
73968|         "SFILTER:  fnl=%08x, '%-*.*S'\n",
73969|         Info->CreationTime,
73970|         Info->LastAccessTime,
73971|         Info->LastWriteTime,
73972|         Info->ChangeTime,
73973|         Info->NextEntryOffset,
73974|         Info->FileIndex,
73975|         Info->EndOfFile,
73976|         Info->AllocationSize,
73977|         Info->FileAttributes,
73978|         Info->FileNameLength /
    | sizeof(WCHAR),Info->FileNameLength /
    | sizeof(WCHAR),Info->FileNameLength / sizeof(WCHAR),
73979|         Info->FileName
73980|     ));
73981|         break;
73982|     }
73983|     case FileBothDirectoryInformation : {
73984|         PFILE_BOTH_DIR_INFORMATION
    | Info=(PFILE_BOTH_DIR_INFORMATION)Irp->AssociatedIrp.Syst
    | emBuffer;
73985|         Debug(DEBUG_SFILTER,("SFILTER: Get
    | FileBothDirectoryInformation for PSM file
    | %08x\n",IrpSp->FileObject));
73986|         Debug(DEBUG_SFILTER,("SFILTER:  Cr=%l64x,
    | A=%l64x, W=%l64x, Ch=%l64x\n"
73987|         "SFILTER:  NEO=%08x, FI=%08x,
    | eof=%l64x, alloc=%l64x, attr=%08x\n"
73988|         "SFILTER:  fnl=%08x, '%-*.*S'\n"
73989|         "SFILTER:  snl=%08x, '%-*.*S'\n",
73990|         Info->CreationTime,

```

```

73991|          Info->LastAccessTime,
73992|          Info->LastWriteTime,
73993|          Info->ChangeTime,
73994|          Info->NextEntryOffset,
73995|          Info->FileIndex,
73996|          Info->EndOfFile,
73997|          Info->AllocationSize,
73998|          Info->FileAttributes,
73999|          Info->FileNameLength /
      | sizeof(WCHAR),Info->FileNameLength /
      | sizeof(WCHAR),Info->FileNameLength / sizeof(WCHAR),
74000|          Info->FileName,
74001|          Info->ShortNameLength /
      | sizeof(WCHAR),Info->ShortNameLength /
      | sizeof(WCHAR),Info->ShortNameLength / sizeof(WCHAR),
74002|          Info->ShortName
74003|      ));
74004|      break;
74005|  }
74006|  case FileBasicInformation : {
74007|      PFILE_BASIC_INFORMATION
      | Basic=(PFILE_BASIC_INFORMATION)Irp->AssociatedIrp.System
      | Buffer;
74008|      Debug(DEBUG_SFILTER,("SFILTER: Get
      | FileBasicInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74009|      Debug(DEBUG_SFILTER,("SFILTER:  Cr=%l64x,
      | A=%l64x, W=%l64x, Ch=%l64x, Attr=%08x\n",
74010|          Basic->CreationTime,
74011|          Basic->LastAccessTime,
74012|          Basic->LastWriteTime,
74013|          Basic->ChangeTime,
74014|          Basic->FileAttributes
74015|      ));
74016|      break;
74017|  }
74018|  case FileStandardInformation : {
74019|      PFILE_STANDARD_INFORMATION
      | Info=(PFILE_STANDARD_INFORMATION)Irp->AssociatedIrp.Syst
      | emBuffer;
74020|      Debug(DEBUG_SFILTER,("SFILTER: Get
      | FileStandardInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74021|      Debug(DEBUG_SFILTER,("SFILTER:
      | Alloc=%l64x, eof=%l64x, nl=%d, dp=%d, dir=%d\n",
74022|          Info->AllocationSize,
74023|          Info->EndOfFile,
74024|          Info->NumberOfLinks,
74025|          Info->DeletePending,
74026|          Info->Directory

```

```

74027|         ));
74028|         break;
74029|     }
74030|     case FileInternalInformation : {
74031|         PFILE_INTERNAL_INFORMATION
74032|         | Info=(PFILE_INTERNAL_INFORMATION)Irp->AssociatedIrp.Syst
74033|         | emBuffer;
74034|         Debug(DEBUG_SFILTER,("SFILTER: Get
74035|         | FileInternalInformation for PSM file
74036|         | %08x\n",IrpSp->FileObject));
74037|         Debug(DEBUG_SFILTER,("SFILTER:
74038|         | index=%0164x\n",
74039|         | Info->IndexNumber
74040|         ));
74041|         break;
74042|     }
74043|     case FileEaInformation : {
74044|         Debug(DEBUG_SFILTER,("SFILTER:
74045|         | FileEaInformation FilePositionInformation for PSM file
74046|         | %08x\n",IrpSp->FileObject));
74047|         break;
74048|     }
74049|     case FileAccessInformation : {
74050|         PFILE_ACCESS_INFORMATION
74051|         | Info=(PFILE_ACCESS_INFORMATION)Irp->AssociatedIrp.System
74052|         | Buffer;
74053|         Debug(DEBUG_SFILTER,("SFILTER: Get
74054|         | FileAccessInformation for PSM file
74055|         | %08x\n",IrpSp->FileObject));
74056|         Debug(DEBUG_SFILTER,("SFILTER:
74057|         | access=%08x\n",
74058|         | Info->AccessFlags
74059|         ));
74060|         break;
74061|     }
74062|     case FileNameInformation : {
74063|         PFILE_NAME_INFORMATION
74064|         | Info=(PFILE_NAME_INFORMATION)Irp->AssociatedIrp.SystemBu
74065|         | ffer;
74066|         Debug(DEBUG_SFILTER,("SFILTER:
74067|         | FileNameInformation FilePositionInformation for PSM
74068|         | file %08x\n",IrpSp->FileObject));
74069|         Debug(DEBUG_SFILTER,("SFILTER: fnl=%d,
74070|         | fn='%-*.*S'\n",
74071|         | Info->FileNameLength /
74072|         | sizeof(WCHAR),Info->FileNameLength /
74073|         | sizeof(WCHAR),Info->FileNameLength / sizeof(WCHAR),
74074|         | Info->FileName
74075|         ));
74076|         break;

```

```

74058|     }
74059|     case FileRenameInformation : {
74060|         PFILE_RENAME_INFORMATION
74061|         | Ren=(PFILE_RENAME_INFORMATION)Irp->AssociatedIrp.SystemB
74062|         | uffer;
74063|         Debug(DEBUG_SFILTER,("SFILTER: Get
74064|         | FileRenameInformation for PSM file
74065|         | %08x\n",IrpSp->FileObject));
74066|         Debug(DEBUG_SFILTER,("SFILTER: rie=%d,
74067|         | rd=%08x, fnl=%d, fn='%-*.*S'\n",
74068|         | Ren->ReplaceIfExists,
74069|         | Ren->RootDirectory,
74070|         | Ren->FileNameLength /
74071|         | sizeof(WCHAR),Ren->FileNameLength /
74072|         | sizeof(WCHAR),Ren->FileNameLength / sizeof(WCHAR),
74073|         | Ren->FileName
74074|         | ));
74075|         break;
74076|     }
74077|     case FileLinkInformation : {
74078|         PFILE_LINK_INFORMATION
74079|         | Ren=(PFILE_LINK_INFORMATION)Irp->AssociatedIrp.SystemBuf
74080|         | fer;
74081|         Debug(DEBUG_SFILTER,("SFILTER: Get
74082|         | FileLinkInformation for PSM file
74083|         | %08x\n",IrpSp->FileObject));
74084|         Debug(DEBUG_SFILTER,("SFILTER: rie=%d,
74085|         | rd=%08x, fnl=%d, fn='%-*.*S'\n",
74086|         | Ren->ReplaceIfExists,
74087|         | Ren->RootDirectory,
74088|         | Ren->FileNameLength /
74089|         | sizeof(WCHAR),Ren->FileNameLength /
74090|         | sizeof(WCHAR),Ren->FileNameLength / sizeof(WCHAR),
74091|         | Ren->FileName
74092|         | ));
74093|         break;
74094|     }
74095|     case FileNamesInformation : {
74096|         PFILE_NAMES_INFORMATION
74097|         | Info=(PFILE_NAMES_INFORMATION)Irp->AssociatedIrp.SystemB
74098|         | uffer;
74099|         Debug(DEBUG_SFILTER,("SFILTER:
74100|         | FileNamesInformation FilePositionInformation for PSM
74101|         | file %08x\n",IrpSp->FileObject));
74102|         break;
74103|     }
74104|     case FileDispositionInformation : {
74105|         PFILE_DISPOSITION_INFORMATION
74106|         | Info=(PFILE_DISPOSITION_INFORMATION)Irp->AssociatedIrp.S
74107|         | ystemBuffer;

```

```

74088|         Debug(DEBUG_SFILTER,("SFILTER: Get
| FileDispositionInformation for PSM file
| %08x\n",IrpSp->FileObject));
74089|         Debug(DEBUG_SFILTER,("SFILTER:
| Disp=%d\n",
74090|             Info->DeleteFile
74091|             ));
74092|         break;
74093|     }
74094|     case FilePositionInformation : {
74095|         PFILE_POSITION_INFORMATION
| Pos=(PFILE_POSITION_INFORMATION)Irp->AssociatedIrp.Syste
| mBuffer;
74096|         Debug(DEBUG_SFILTER,("SFILTER: Get
| FilePositionInformation for PSM file
| %08x\n",IrpSp->FileObject));
74097|         Debug(DEBUG_SFILTER,("SFILTER:
| Pos=%l64x\n",Pos->CurrentByteOffset));
74098|         break;
74099|     }
74100|     case FileFullEaInformation : {
74101|         PFILE_FULL_EA_INFORMATION
| Info=(PFILE_FULL_EA_INFORMATION)Irp->AssociatedIrp.Syste
| mBuffer;
74102|         Debug(DEBUG_SFILTER,("SFILTER: Get
| FileFullEaInformation for PSM file
| %08x\n",IrpSp->FileObject));
74103|         break;
74104|     }
74105|     case FileModeInformation : {
74106|         PFILE_MODE_INFORMATION
| Info=(PFILE_MODE_INFORMATION)Irp->AssociatedIrp.SystemBu
| ffer;
74107|         Debug(DEBUG_SFILTER,("SFILTER:
| FileModeInformation FilePositionInformation for PSM
| file %08x\n",IrpSp->FileObject));
74108|         Debug(DEBUG_SFILTER,("SFILTER:
| Mode=%08x\n",
74109|             Info->Mode
74110|             ));
74111|         break;
74112|     }
74113|     case FileAlignmentInformation : {
74114|         PFILE_ALIGNMENT_INFORMATION
| Info=(PFILE_ALIGNMENT_INFORMATION)Irp->AssociatedIrp.Sys
| temBuffer;
74115|         Debug(DEBUG_SFILTER,("SFILTER: Get
| FileAlignmentInformation for PSM file
| %08x\n",IrpSp->FileObject));
74116|         Debug(DEBUG_SFILTER,("SFILTER:

```

```

    | Align=%d\n",
74117|         Info->AlignmentRequirement
74118|     ));
74119|         break;
74120|     }
74121|     case FileAllInformation : {
74122|         PFILE_ALL_INFORMATION
    | All=(PFILE_ALL_INFORMATION)Irp->AssociatedIrp.SystemBuff
    | er;
74123|         Debug(DEBUG_SFILTER,("SFILTER:
    | FileAllInformation FilePositionInformation for PSM file
    | %08x\n",IrpSp->FileObject));
74124|         Debug(DEBUG_SFILTER,("SFILTER:  Cr=%I64x,
    | A=%I64x, W=%I64x, Ch=%I64x, Attr=%08x\n",
74125|             All->BasicInformation.CreationTime,
74126|             All->BasicInformation.LastAccessTime,
74127|             All->BasicInformation.LastWriteTime,
74128|             All->BasicInformation.ChangeTime,
74129|             All->BasicInformation.FileAttributes
74130|         ));
74131|         Debug(DEBUG_SFILTER,("SFILTER:
    | Alloc=%I64x, eof=%I64x, nl=%d, dp=%d, dir=%d\n",
74132|             All->StandardInformation.AllocationSize,
74133|             All->StandardInformation.EndOfFile,
74134|             All->StandardInformation.NumberOfLinks,
74135|             All->StandardInformation.DeletePending,
74136|             All->StandardInformation.Directory
74137|         ));
74138|         Debug(DEBUG_SFILTER,("SFILTER:
    | index=%I64x\n",All->InternalInformation.IndexNumber));
74139|         // ea
74140|         Debug(DEBUG_SFILTER,("SFILTER:
    | access=%08x\n",All->AccessInformation.AccessFlags));
74141|         Debug(DEBUG_SFILTER,("SFILTER:
    | Pos=%I64x\n",All->PositionInformation.CurrentByteOffset
    | ));
74142|         Debug(DEBUG_SFILTER,("SFILTER:
    | Mode=%08x\n",All->ModeInformation.Mode));
74143|         Debug(DEBUG_SFILTER,("SFILTER:
    | Align=%d\n",All->AlignmentInformation.AlignmentRequireme
    | nt));
74144|         Debug(DEBUG_SFILTER,("SFILTER:  fnl=%d,
    | fn='%-*.*S'\n",
74145|             All->NameInformation.FileNameLength /
    | sizeof(WCHAR),All->NameInformation.FileNameLength /
    | sizeof(WCHAR),All->NameInformation.FileNameLength /
    | sizeof(WCHAR),
74146|             All->NameInformation.FileName
74147|         ));

```

```

74148|
74149|     break;
74150| }
74151|     case FileAllocationInformation : {
74152|         PFILE_ALLOCATION_INFORMATION
74153|         | Alloc=(PFILE_ALLOCATION_INFORMATION)Irp->AssociatedIrp.S
74154|         | ystemBuffer;
74155|         Debug(DEBUG_SFILTER,("SFILTER: Get
74156|         | FileAllocationInformation for PSM file
74157|         | %08x\n",IrpSp->FileObject));
74158|         Debug(DEBUG_SFILTER,("SFILTER:
74159|         | alloc=%l64x\n",Alloc->AllocationSize));
74160|         break;
74161|     }
74162|     case FileEndOfFileInformation : {
74163|         PFILE_END_OF_FILE_INFORMATION
74164|         | End=(PFILE_END_OF_FILE_INFORMATION)Irp->AssociatedIrp.Sy
74165|         | stemBuffer;
74166|         Debug(DEBUG_SFILTER,("SFILTER: Get
74167|         | FileEndOfFileInformation for PSM file
74168|         | %08x\n",IrpSp->FileObject));
74169|         Debug(DEBUG_SFILTER,("SFILTER:
74170|         | %l64x\n",End->EndOfFile));
74171|         break;
74172|     }
74173|     case FileAlternateNameInformation : {
74174|         Debug(DEBUG_SFILTER,("SFILTER: Get
74175|         | FileAlternateNameInformation for PSM file
74176|         | %08x\n",IrpSp->FileObject));
74177|         break;
74178|     }
74179|     case FileStreamInformation : {
74180|         PFILE_STREAM_INFORMATION
74181|         | Info=(PFILE_STREAM_INFORMATION)Irp->AssociatedIrp.System
74182|         | Buffer;
74183|         Debug(DEBUG_SFILTER,("SFILTER: Get
74184|         | FileStreamInformation for PSM file
74185|         | %08x\n",IrpSp->FileObject));
74186|         break;
74187|     }
74188|     case FilePipeInformation : {
74189|         PFILE_PIPE_INFORMATION
74190|         | Info=(PFILE_PIPE_INFORMATION)Irp->AssociatedIrp.SystemBu
74191|         | ffer;
74192|         Debug(DEBUG_SFILTER,("SFILTER:
74193|         | FilePipeInformation FilePositionInformation for PSM
74194|         | file %08x\n",IrpSp->FileObject));
74195|         break;
74196|     }
74197|     case FilePipeLocalInformation : {

```

```

74178|         PFILE_PIPE_LOCAL_INFORMATION
      | Info=(PFILE_PIPE_LOCAL_INFORMATION)Irp->AssociatedIrp.Sy
      | stemBuffer;
74179|         Debug(DEBUG_SFILTER,("SFILTER: Get
      | FilePipeLocalInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74180|         break;
74181|     }
74182|     case FilePipeRemoteInformation : {
74183|         PFILE_PIPE_REMOTE_INFORMATION
      | Info=(PFILE_PIPE_REMOTE_INFORMATION)Irp->AssociatedIrp.S
      | ystemBuffer;
74184|         Debug(DEBUG_SFILTER,("SFILTER: Get
      | FilePipeRemoteInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74185|         break;
74186|     }
74187|     case FileMailslotQueryInformation : {
74188|         Debug(DEBUG_SFILTER,("SFILTER: Get
      | FileMailslotQueryInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74189|         break;
74190|     }
74191|     case FileMailslotSetInformation : {
74192|         Debug(DEBUG_SFILTER,("SFILTER: Get
      | FileMailslotSetInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74193|         break;
74194|     }
74195|     case FileCompressionInformation : {
74196|         Debug(DEBUG_SFILTER,("SFILTER: Get
      | FileCompressionInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74197|         break;
74198|     }
74199|     case FileObjectIdInformation : {
74200|         Debug(DEBUG_SFILTER,("SFILTER: Get
      | FileObjectIdInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74201|         break;
74202|     }
74203|     case FileCompletionInformation : {
74204|         Debug(DEBUG_SFILTER,("SFILTER: Get
      | FileCompletionInformation for PSM file
      | %08x\n",IrpSp->FileObject));
74205|         break;
74206|     }
74207|     case FileMoveClusterInformation : {
74208|         Debug(DEBUG_SFILTER,("SFILTER: Get
      | FileMoveClusterInformation for PSM file

```



```

    | %08x\n", IrpSp->FileObject));
74209|         break;
74210|     }
74211|     case FileQuotaInformation : {
74212|         Debug(DEBUG_SFILTER, ("SFILTER:
    | FileQuotaInformation FilePositionInformation for PSM
    | file %08x\n", IrpSp->FileObject));
74213|         break;
74214|     }
74215|     case FileReparsePointInformation : {
74216|         Debug(DEBUG_SFILTER, ("SFILTER: Get
    | FileReparsePointInformation for PSM file
    | %08x\n", IrpSp->FileObject));
74217|         break;
74218|     }
74219|     case FileNetworkOpenInformation : {
74220|         PFILE_NETWORK_OPEN_INFORMATION
    | Info=(PFILE_NETWORK_OPEN_INFORMATION)Irp->AssociatedIrp.
    | SystemBuffer;
74221|         Debug(DEBUG_SFILTER, ("SFILTER: Get
    | FileNetworkOpenInformation for PSM file
    | %08x\n", IrpSp->FileObject));
74222|         Debug(DEBUG_SFILTER, ("SFILTER: Cr=%l64x,
    | A=%l64x, W=%l64x, Ch=%l64x\n"
74223|         "SFILTER: Alloc=%l64x, eof=%l64x,
    | attr=%08x\n",
74224|         Info->CreationTime,
74225|         Info->LastAccessTime,
74226|         Info->LastWriteTime,
74227|         Info->ChangeTime,
74228|         Info->AllocationSize,
74229|         Info->EndOfFile,
74230|         Info->FileAttributes
74231|     ));
74232|
74233|         break;
74234|     }
74235|     case FileAttributeTagInformation : {
74236|         PFILE_ATTRIBUTE_TAG_INFORMATION
    | Info=(PFILE_ATTRIBUTE_TAG_INFORMATION)Irp->AssociatedIrp
    | .SystemBuffer;
74237|         Debug(DEBUG_SFILTER, ("SFILTER: Get
    | FileAttributeTagInformation for PSM file
    | %08x\n", IrpSp->FileObject));
74238|         Debug(DEBUG_SFILTER, ("SFILTER: Attr=%08x,
    | Tag=%08x\n", Info->FileAttributes, Info->ReparseTag));
74239|         break;
74240|     }
74241|     case FileTrackingInformation : {
74242|         Debug(DEBUG_SFILTER, ("SFILTER: Get

```

```

    | FileTrackingInformation for PSM file
    | %08x\n",IrpSp->FileObject));
74243|         break;
74244|     }
74245|
74246| }
74247| return STATUS_SUCCESS;
74248| }
74249|
74250| NTSTATUS
74251| SfFsQueryInformation(
74252|         IN PDEVICE_OBJECT DeviceObject,
74253|         IN PIRP Irp
74254|         )
74255| {
74256|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
74257|
74258|     switch ( PsmGetObjectype(DeviceObject) ) {
74259|         case OBJECT_FILTEREDDISK :
74260|             //Debug(DEBUG_SFILTER,("SFILTER: Query: fd
    | %08x\n",DeviceObject));
74261|             return PSMANFSPassThru( DeviceObject, Irp
    | );
74262|         case OBJECT_FS_FILTER :
74263|             //Debug(DEBUG_SFILTER,("SFILTER: Query: fs
    | %08x\n",DeviceObject));
74264|
74265|
    | if(FileIsPSM(loGetCurrentIrpStackLocation(Irp)->FileObje
    | ct, FALSE)) {
74266|             IoCopyCurrentIrpStackLocationToNext(
    | Irp );
74267|
74268|             IoSetCompletionRoutine(
74269|                 Irp,
74270|
    | QueryInformationCompletionRoutine,
74271|                 NULL, // arg
74272|                 TRUE,
74273|                 FALSE,
74274|                 FALSE
74275|                 );
74276|
74277|             return
    | IoCallDriver(((PFILTERED_EXTENSION)GetDeviceExtension(De
    | viceObject))->TargetDeviceObject, Irp);
74278|         } else {
74279|             // pass it down
74280|             return PSMANFSPassThru( DeviceObject,
    | Irp );

```

```

74281|     }
74282|     case OBJECT_INTERNAL :
74283|     case OBJECT_VIRTUALDISK :
74284|     case OBJECT_FS_OBJECT :
74285|         //Debug(DEBUG_SFILTER,("SFILTER: Dir: Obj
    | %08x\n",DeviceObject));
74286|         break;
74287|     default:
74288|         Status = STATUS_NO_SUCH_DEVICE;
74289|         break;
74290| }
74291|
74292| Irp->IoStatus.Information = 0;
74293| Irp->IoStatus.Status = Status;
74294| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
74295| return Status;
74296| }
74297|
74298| NTSTATUS DeleteAllMaps( PFILTERED_EXTENSION DevExt )
74299| {
74300|     WCHAR *LocalReg;
74301|
74302|     Debug(DEBUG_DICT,("DeleteAllMaps %08x %08x
    | %08x\n",DevExt->Cache.HeaderFile.Direct,DevExt->Cache.In
    | dexFile.Direct,DevExt->Cache.CacheFile.Direct));
74303|     KeEnterCriticalRegion();
74304|     pmAcquireWriterLock (
    | &DevExt->Cache.DirectAccessResource, TRUE );
74305|     DirectAccessFile *oldHeaderDirect =
    | DevExt->Cache.HeaderFile.Direct;
74306|     DirectAccessFile *oldIndexDirect =
    | DevExt->Cache.IndexFile.Direct;
74307|     DirectAccessFile *oldCacheDirect =
    | DevExt->Cache.CacheFile.Direct;
74308|     DevExt->Cache.HeaderFile.Direct = NULL;
74309|     DevExt->Cache.IndexFile.Direct = NULL;
74310|     DevExt->Cache.CacheFile.Direct = NULL;
74311|     pmReleaseWriterLock (
    | &DevExt->Cache.DirectAccessResource );
74312|     KeLeaveCriticalRegion();
74313|
74314|     delete oldHeaderDirect;
74315|     delete oldIndexDirect;
74316|     delete oldCacheDirect;
74317|     oldHeaderDirect = oldIndexDirect = oldCacheDirect =
    | NULL;
74318|
74319|     LocalReg=(WCHAR*)MemAllocateString(256);
74320|     if(LocalReg) {
74321|

```

```

    | RtlCopyMemory(LocalReg,gRegistryPath.Buffer,gRegistryPat
    | h.Length);
74322|     LocalReg[gRegistryPath.Length / 2] = 0;
74323|     wcscat(LocalReg,L"\\");
74324|     wcscat(LocalReg,DevExt->VolumeGuid);
74325|
74326|
    | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg,L"
    | CacheMap");
74327|
    | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg,L"
    | IndexMap");
74328|
    | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg,L"
    | HeaderMap");
74329|
74330| #if 0
74331|     // FIXFIX need to delete from all nodes in
    | cluster, not just this node
74332|
    | wcscpy(LocalReg,L"\\Registry\\Machine\\Cluster\\Persiste
    | ntStorageManager\\");
74333|     wcscat(LocalReg,DevExt->UniqueId);
74334|
    | if(RtlCheckRegistryKey(RTL_REGISTRY_ABSOLUTE,LocalReg)==
    | STATUS_SUCCESS) {
74335|         // delete it
74336|
    | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg,L"
    | CacheMap");
74337|
    | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg,L"
    | IndexMap");
74338|
    | RtlDeleteRegistryValue(RTL_REGISTRY_ABSOLUTE,LocalReg,L"
    | HeaderMap");
74339|     }
74340| #endif
74341|
74342|     MemFreeString(LocalReg);
74343| }
74344| return STATUS_SUCCESS;
74345| }
74346|
74347| NTSTATUS
74348| SfFsSetInformation(
74349|     IN PDEVICE_OBJECT DeviceObject,
74350|     IN PIRP Irp
74351| )
74352| {

```

```

74353| NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
74354|
74355| switch ( PsmGetObjectType(DeviceObject) ) {
74356|     case OBJECT_FILTEREDDISK :
74357|         //Debug(DEBUG_SFILTER,("SFILTER: SetInfo:
| fd %08x\n",DeviceObject));
74358|         return PSMANFSPassThru( DeviceObject, Irp
| );
74359|     case OBJECT_FS_FILTER : {
74360|         PFS_FILTER_EXTENSION DevExt =
| (PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceObject);
74361|         PIO_STACK_LOCATION IrpSp =
| IoGetCurrentIrpStackLocation( Irp );
74362|         ASSERT(DevExt->ObjectType ==
| OBJECT_FS_FILTER);
74363|
74364|         //Debug(DEBUG_SFILTER,("SFILTER: Dir:
| fs %08x\n",DeviceObject));
74365|         if ( (DevExt->Virtual) &&
| (DevExt->PSMStorageObject) ) {
74366|             PVDISK_EXTENSION
| VDiskExt=(PVDISK_EXTENSION)GetDeviceExtension(DevExt->PS
| MStorageObject);
74367|             if ( VDiskExt->SnapShot ) {
74368|                 pPersistentDictionary
| dictionary =
| (pPersistentDictionary)VDiskExt->SnapShot->Dictionary;
74369|                 ASSERT ( dictionary != NULL );
74370|                 if ( dictionary != NULL ) {
74371|                     if (
| dictionary->IsReadOnly() ) {
74372|                         if (
| FileIsReadOnly(IrpSp->FileObject) ) {
74373|                             // virtual volume
| is marked readonly, deny this write
74374|
| Debug(DEBUG_SFILTER,("SFILTER: %08x attempted on read
| only
| volume\n",IrpSp->Parameters.SetFile.FileInformationClass
| ));
74375|
| Irp->IoStatus.Status = STATUS_MEDIA_WRITE_PROTECTED;
74376|
| Irp->IoStatus.Information = 0;
74377|                 IoCompleteRequest(
| Irp, IO_NO_INCREMENT );
74378|                 return
| STATUS_MEDIA_WRITE_PROTECTED;
74379|             }
74380|         } else {

```

```

74381|                // not a read-only
| snapshot. Need to check cache usage...
74382|                if ( !(gVDiskIOHandling
| & PSM_VDISK_FLAG_CACHE_FULL_DELETE_SS) ) {
74383|                    if (
| dictionary->IsCacheWarningThresholdReached() ) {
74384|                        if (
| FileIsOpenOnSnapShot(IrpSp->FileObject,FALSE) ) {
74385|                            // Fail the
| write to the snapshot because too much cache is in
| use...
74386|                            NTSTATUS
| Status = Irp->IoStatus.Status = STATUS_DISK_FULL;
74387|                            Irp->IoStatus.Information = 0;
74388|                            IoCompleteRequest( Irp, IO_NO_INCREMENT );
74389|                            Debug(DEBUG_SFILTER,("SfFsSetInformation: Reporting
| STATUS_DISK_FULL; DevExt=%08x, VDiskExt=%08x\n",DevExt,
| VDiskExt));
74390|                            return
| Status;
74391|                        }
74392|                    }
74393|                }
74394|            }
74395|        }
74396|    }
74397|    } else
74398|        if ( !(DevExt->Virtual)) &&
| (DevExt->PSMStorageObject) ) {
74399|            PFILTERED_EXTENSION
| FiltExt=(PFILTERED_EXTENSION)GetDeviceExtension(DevExt->
| PSMStorageObject);
74400|
74401|            if(FileIsPSM(IrpSp->FileObject,FALSE)) {
74402|                // filtered disk
74403|
| switch(IrpSp->Parameters.SetFile.FileInformationClass)
| {
74404|
74405|                case
| FileAllocationInformation: {
74406|
| PFILE_ALLOCATION_INFORMATION
| Alloc=(PFILE_ALLOCATION_INFORMATION)Irp->AssociatedIrp.S
| ystemBuffer;
74407|

```

```

    | Debug(DEBUG_SFILTER,("SFILTER: Set
    | FileAllocationInformation for PSM file
    | %08x\n",IrpSp->FileObject));
74408|
    | Debug(DEBUG_SFILTER,("SFILTER:
    | alloc=%I64x\n",Alloc->AllocationSize));
74409|                break;
74410|                }
74411|                case FileLinkInformation: {
74412|                PFILE_LINK_INFORMATION
    | Ren=(PFILE_LINK_INFORMATION)Irp->AssociatedIrp.SystemBuf
    | fer;
74413|
    | Debug(DEBUG_SFILTER,("SFILTER: Set FileLinkInformation
    | for PSM file %08x\n",IrpSp->FileObject));
74414|
    | Debug(DEBUG_SFILTER,("SFILTER:  rie=%d, rd=%08x,
    | fnl=%d, fn='%-*.*S'\n",
74415|
    | Ren->ReplacelfExists,
74416|                Ren->RootDirectory,
74417|                Ren->FileNameLength
    | / sizeof(WCHAR),Ren->FileNameLength /
    | sizeof(WCHAR),Ren->FileNameLength / sizeof(WCHAR),
74418|                Ren->FileName
74419|
74420|                ));
74421|                break;
74422|                }
74423|                case
    | FilePositionInformation: {
74424|
    | PFILE_POSITION_INFORMATION
    | Pos=(PFILE_POSITION_INFORMATION)Irp->AssociatedIrp.Syste
    | mBuffer;
74425|
74426|
    | Debug(DEBUG_SFILTER,("SFILTER: Set
    | FilePositionInformation for PSM file
    | %08x\n",IrpSp->FileObject));
74427|
    | Debug(DEBUG_SFILTER,("SFILTER:
    | Pos=%I64x\n",Pos->CurrentByteOffset));
74428|                // changes file
    | position, we dont care.
74429|                break;
74430|                }
74431|                case FileBasicInformation:
    | {
74432|                PFILE_BASIC_INFORMATION

```

```

    | Basic=(PFILE_BASIC_INFORMATION)Irp->AssociatedIrp.System
    | Buffer;
74433|
    | Debug(DEBUG_SFILTER,("SFILTER: Set FileBasicInformation
    | for PSM file %08x\n",IrpSp->FileObject));
74434|
    | Debug(DEBUG_SFILTER,("SFILTER:  Cr=%I64x, A=%I64x,
    | W=%I64x, Ch=%I64x, Attr=%08x\n",
74435|
    | Basic->CreationTime,
74436|
    | Basic->LastAccessTime,
74437|
    | Basic->LastWriteTime,
74438|                Basic->ChangeTime,
74439|
    | Basic->FileAttributes
74440|                ));
74441|                // changes times and
    | attributes of files.. we dont care..
74442|                break;
74443|        }
74444|        case FileRenameInformation:
    | {
74445|
    | PFILE_RENAME_INFORMATION
    | Ren=(PFILE_RENAME_INFORMATION)Irp->AssociatedIrp.SystemB
    | uffer;
74446|
    | Debug(DEBUG_SFILTER,("SFILTER: Set
    | FileRenameInformation for PSM file
    | %08x\n",IrpSp->FileObject));
74447|
    | Debug(DEBUG_SFILTER,("SFILTER:  rie=%d, rd=%08x,
    | fnl=%d, fn='%-*.*S'\n",
74448|
    | Ren->ReplaceIfExists,
74449|                Ren->RootDirectory,
74450|                Ren->FileNameLength
    | / sizeof(WCHAR),Ren->FileNameLength /
    | sizeof(WCHAR),Ren->FileNameLength / sizeof(WCHAR),
74451|                Ren->FileName
74452|
74453|                ));
74454|                // delete our maps,
    | this is probably coming in from the files being moved
    | to the
74455|                // recycle bin
74456|
    | ASSERT(FiltExt->IsMounted);

```



```

74457|                if(FiltExt->IsMounted)
74458|                | {
74459|                | DeleteAllMaps(FiltExt);
74460|                | NotifyUserModeOfRegChangeEvent(FiltExt);
74461|                | } // is mounted
74462|                break;
74463|                }
74464|                case
74465|                | FileDispositionInformation: {
74466|                | PFILE_DISPOSITION_INFORMATION
74467|                | Disp=(PFILE_DISPOSITION_INFORMATION)Irp->AssociatedIrp.S
74468|                | ystemBuffer;
74469|                | Debug(DEBUG_SFILTER,("SFILTER: Set
74470|                | FileDispositionInformation for PSM file
74471|                | %08x\n",IrpSp->FileObject));
74472|                | Debug(DEBUG_SFILTER,("SFILTER:
74473|                | %d\n",Disp->DeleteFile));
74474|                // deletes the file
74475|                // we need to get rid
74476|                | of our direct io maps in the devext, local registry,
74477|                | and cluster registry
74478|                // cant do things to
74479|                | files if in direct mode
74480|                | ASSERT(FiltExt->IsMounted);
74481|                if(FiltExt->IsMounted)
74482|                | {
74483|                | DeleteAllMaps(FiltExt);
74484|                | NotifyUserModeOfRegChangeEvent(FiltExt);
74485|                | } // is mounted
74486|                break;
74487|                }
74488|                case
74489|                | FileEndOfFileInformation: {
74490|                | PFILE_END_OF_FILE_INFORMATION
74491|                | End=(PFILE_END_OF_FILE_INFORMATION)Irp->AssociatedIrp.Sy
74492|                | stemBuffer;
74493|                | Debug(DEBUG_SFILTER,("SFILTER: Set
74494|                | FileEndOfFileInformation for PSM file
74495|                | %08x\n",IrpSp->FileObject));
74496|

```

```

    | Debug(DEBUG_SFILTER,("SFILTER:
    | %!64x\n",End->EndOfFile));
74481|          // truncates or extends
    | the file, we need to update our directio maps, local
    | registry, and cluster registry
74482|
    | ASSERT(FiltExt->DoDirectIO==FALSE);
74483|
    | ASSERT(FiltExt->IsMounted);
74484|          if(FiltExt->IsMounted)
    | {
74485|          NTSTATUS Status;
74486|
    | DeleteAllMaps(FiltExt);
74487|
74488|          Status =
    | PSMANForwardIrpSynchronous(DeviceObject,Irp);
74489|
    | if(NT_SUCCESS(Status)) {
74490|
    | PersistentDictionary::StoreClustersOfFiles(FiltExt->Devi
    | ceObject);
74491|
    | PersistentDictionary::RetrieveDirectIOMaps(FiltExt->Devi
    | ceObject,FALSE);
74492|
    | NotifyUserModeOfRegChangeEvent(FiltExt);
74493|          }
74494|          // finish the
    | request
74495|
    | Irp->IoStatus.Status = Status;
74496|
    | IoCompleteRequest(Irp,IO_DISK_INCREMENT);
74497|          return Status;
74498|          }
74499|          break;
74500|          }
74501|          default:
74502|
    | Debug(DEBUG_SFILTER,("SFILTER: Unknown file class %08x
    | for PSM file
    | %08x\n",IrpSp->Parameters.SetFile.FileInformationClass,I
    | rpSp->FileObject));
74503|          break;
74504|          } // switch
74505|          } // if psm file
74506|          } // if
74507|          return PSMANFSPassThru( DeviceObject,
    | Irp );

```

```

74508|         } // fsfilter
74509|     case OBJECT_INTERNAL :
74510|     case OBJECT_VIRTUALDISK :
74511|     case OBJECT_FS_OBJECT :
74512|         //Debug(DEBUG_SFILTER,("SFILTER: Dir: obj
| %08x\n",DeviceObject));
74513|         break;
74514|     default:
74515|         Status = STATUS_NO_SUCH_DEVICE;
74516|         break;
74517| }
74518|
74519| Irp->IoStatus.Information = 0;
74520| Irp->IoStatus.Status = Status;
74521| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
74522| return Status;
74523| }
74524|
74525| NTSTATUS
74526| SfFsDirectoryControl(
74527|     IN PDEVICE_OBJECT DeviceObject,
74528|     IN PIRP Irp
74529| )
74530| {
74531|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
74532|
74533|     switch ( PsmGetObjectType(DeviceObject) ) {
74534|     case OBJECT_FILTEREDDISK :
74535|         //Debug(DEBUG_SFILTER,("SFILTER: Dir: fd
| %08x\n",DeviceObject));
74536|         return PSMANFSPassThru( DeviceObject, Irp
| );
74537|     case OBJECT_FS_FILTER :
74538|         // not currently doing anything
74539|         //Debug(DEBUG_SFILTER,("SFILTER: Dir: fs
| %08x\n",DeviceObject));
74540|         return PSMANFSPassThru( DeviceObject, Irp
| );
74541|     case OBJECT_INTERNAL :
74542|     case OBJECT_VIRTUALDISK :
74543|     case OBJECT_FS_OBJECT :
74544|         //Debug(DEBUG_SFILTER,("SFILTER: Dir: obj
| %08x\n",DeviceObject));
74545|         break;
74546|     default:
74547|         Status = STATUS_NO_SUCH_DEVICE;
74548|         break;
74549| }
74550|
74551| Irp->IoStatus.Information = 0;

```

```

74552|     Irp->IoStatus.Status = Status;
74553|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
74554|     return Status;
74555| }
74556|
74557|
74558| NTSTATUS
74559| SfFsWrite(
74560|     PDEVICE_OBJECT DeviceObject,
74561|     PIRP Irp
74562| )
74563| {
74564|     PFS_FILTER_EXTENSION DevExt =
74565|         (PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceObject);
74566|     ASSERT(DevExt->ObjectType == OBJECT_FS_FILTER);
74567|
74568|     if (
74569|         | PsmGetObjectType(DeviceObject)!=OBJECT_FS_FILTER ) {
74570|         Debug(DEBUG_SFILTER,("SFILTER: Write: Not fs
74571|         | object\n"));
74572|         return FALSE;
74573|     }
74574|
74575|     // ebug(DEBUG_SFILTER,("SFILTER: Write\n"));
74576|     if ( (DevExt->Virtual) &&
74577|         | (DevExt->PSMStorageObject) ) {
74578|         PVDISK_EXTENSION
74579|         | VDiskExt=(PVDISK_EXTENSION)GetDeviceExtension(DevExt->PS
74580|         | MStorageObject);
74581|
74582|         if ( VDiskExt->SnapShot != NULL ) {
74583|             pPersistentDictionary dictionary =
74584|             | (pPersistentDictionary) VDiskExt->SnapShot->Dictionary;
74585|             ASSERT ( dictionary != NULL );
74586|             if ( dictionary != NULL ) {
74587|                 PIO_STACK_LOCATION IrpSp =
74588|                 | IoGetCurrentIrpStackLocation( Irp );
74589|                 if ( dictionary->IsReadOnly() ) {
74590|                     // we check for this, because NTFS
74591|                     | will send writes down to us
74592|                     // for file objects we have never
74593|                     | seen (its metadata), and if we fail
74594|                     // them, lost delay writes popups
74595|                     | will occur, so only deny writes for
74596|                     // file objects we have actually
74597|                     | seen
74598|                     if (
74599|                         | FileIsReadOnly(IrpSp->FileObject) ) {
74600|                         // virtual volume is marked

```

```

    | readonly, deny this write
74589|         Debug(DEBUG_SFILTER,("SFILTER:
    | Write: Write to readonly volume\n"));
74590|         Irp->IoStatus.Information = 0;
74591|         Irp->IoStatus.Status =
    | STATUS_MEDIA_WRITE_PROTECTED;
74592|
    | IoCompleteRequest(Irp,IO_NO_INCREMENT);
74593|         return
    | STATUS_MEDIA_WRITE_PROTECTED;
74594|     } else {
74595|
    | //Debug(DEBUG_SFILTER,("SFILTER: Write: Write to
    | readonly volume for object we dont know about
    | %08x\n",IrpSp->FileObject));
74596|     }
74597| } else {
74598|     // Not a read-only snapshot. Need
    | to check cache usage...
74599|     if ( !(gVDiskIOHandling &
    | PSM_VDISK_FLAG_CACHE_FULL_DELETE_SS) ) {
74600|         if (
    | dictionary->IsCacheWarningThresholdReached() ) {
74601|             if (
    | FileIsOpenOnSnapShot(IrpSp->FileObject,FALSE) ) {
74602|                 // Fail the write to
    | the snapshot because too much cache is in use...
74603|                 NTSTATUS Status =
    | Irp->IoStatus.Status = STATUS_DISK_FULL;
74604|
    | Irp->IoStatus.Information = 0;
74605|
    | IoCompleteRequest(Irp,IO_NO_INCREMENT);
74606|
    | Debug(DEBUG_SFILTER,("SfFsWrite: Reporting
    | STATUS_DISK_FULL on virtual volume; DevExt=%08x,
    | VDiskExt=%08x\n",DevExt,VDiskExt));
74607|         return Status;
74608|     }
74609| }
74610| }
74611| }
74612| }
74613| }
74614| } else
74615| if ( (!DevExt->Virtual) &&
    | (DevExt->PSMStorageObject) ) {
74616|     PFILTERED_EXTENSION Ext =
    | (PFILTERED_EXTENSION)GetDeviceExtension(DevExt->PSMStora
    | geObject);

```

```

74617|
74618|    // make sure to keep in sync with
    | write:WriteDevice
74619|    if((Ext->PSMed) &&
    | (Ext->Cache.CacheFullAction)) {
74620|        if ( (!IsCacheFile(DeviceObject,Irp)) && (
    | !File_IsPagingFile(DeviceObject,Irp) )) {
74621|            ULONG CacheThreshold =
    | (ULONG)(((unsigned __int64)Ext->Cache.PSManBitMapSize *
    | Ext->Cache.CacheFullActionPercent) / 100);
74622|            if (
    | Ext->Cache.CurrentCacheFileSize>CacheThreshold ) {
74623|                Debug(DEBUG_SFILTER,"SFILTER:
    | Write: Above cache full action threshold;
    | CurrentCache=%08x,
    | Threshold=%08x\n",Ext->Cache.CurrentCacheFileSize,
    | CacheThreshold));
74624|                switch(Ext->Cache.CacheFullAction)
    | {
74625|                    case CACHE_ACTION_DENY_WRITES :
74626|                        // try and keep always keep
74627|
    | if(AreThereAlwaysKeepSnapShots()) {
74628|                            NTSTATUS Status =
    | Irp->IoStatus.Status = STATUS_DISK_FULL;
74629|
    | Irp->IoStatus.Information = 0;
74630|
    | IoCompleteRequest(Irp,IO_NO_INCREMENT);
74631|
    | Debug(DEBUG_SFILTER,"SfFsWrite: Reporting
    | STATUS_DISK_FULL on filtered volume;
    | DevExt=%08x\n",DevExt));
74632|                            return Status;
74633|                        }
74634|                        break;
74635|                    case CACHE_ACTION_BSOD :
74636|
    | PSManBugCheck(SB_BUG_WRITE_FILE,SB_CACHE_FULL,Ext->Cache
    | .CurrentCacheFileSize,CacheThreshold,Ext->Cache.CacheFul
    | lActionPercent);
74637|                            break;
74638|                    case
    | CACHE_ACTION_DELETE_ALWAYS_KEEPS:
74639|                            break;
74640|                    default:
74641|                            break;
74642|                }
74643|            } // if above threshold
74644|        } // not cache or pagefile

```

```

74645|     } // if psmed
74646| } // if not virtual
74647| return PSMANFSPassThru(DeviceObject,Irp);
74648| }
74649|
74650| ULONG EnableWritesToNewFiles()
74651| {
74652|     return NewFileWritesAllowed++;
74653| }
74654|
74655| ULONG DisableWritesToNewFiles()
74656| {
74657|     if(NewFileWritesAllowed) {
74658|         return NewFileWritesAllowed--;
74659|     } else {
74660| #ifdef DEBUG
74661|         PVOID CallerAddress=0, CallersCallerAddress=0;
74662|         RtlGetCallersAddress (&CallerAddress,
74663|             | &CallersCallerAddress);
74664|         Debug(DEBUG_DICT,("SFILTER:
74665|             | DisableWritesToNewFiles: Caller=%08x,
74666|             | grandpa=%08x\n",CallerAddress,CallersCallerAddress));
74667|         DbgBreakPoint();
74668| #endif
74669|         return 0;
74670|     }
74671| }
74672| }
74673|
74674| char *File_GetFSCTLFunctionName( ULONG Minor, ULONG
74675|     | FsCtl )
74676| {
74677|     switch ( Minor ) {
74678|         case IRP_MN_USER_FS_REQUEST : {
74679|             switch ( FsCtl ) {
74680|                 case FSCTL_REQUEST_OPLOCK_LEVEL_1 :
74681|                     | return "FSCTL_REQUEST_OPLOCK_LEVEL_1";
74682|                 case FSCTL_REQUEST_OPLOCK_LEVEL_2 :
74683|                     | return "FSCTL_REQUEST_OPLOCK_LEVEL_2";
74684|                 case FSCTL_REQUEST_BATCH_OPLOCK :
74685|                     | return "FSCTL_REQUEST_BATCH_OPLOCK";
74686|                 case FSCTL_OPLOCK_BREAK_ACKNOWLEDGE
74687|                     | : return "FSCTL_OPLOCK_BREAK_ACKNOWLEDGE";
74688|                 case FSCTL_OPBATCH_ACK_CLOSE_PENDING :
74689|                     | return "FSCTL_OPBATCH_ACK_CLOSE_PENDING";
74690|                 case FSCTL_OPLOCK_BREAK_NOTIFY : return
74691|                     | "FSCTL_OPLOCK_BREAK_NOTIFY";
74692|                 case FSCTL_LOCK_VOLUME : return
74693|                     | "FSCTL_LOCK_VOLUME";
74694|                 case FSCTL_UNLOCK_VOLUME : return
74695|                     | "FSCTL_UNLOCK_VOLUME";

```

```

74683|         case FSCTL_DISMOUNT_VOLUME : return
| "FSCTL_DISMOUNT_VOLUME";
74684|         case FSCTL_IS_VOLUME_MOUNTED : return
| "FSCTL_IS_VOLUME_MOUNTED";
74685|         case FSCTL_IS_PATHNAME_VALID : return
| "FSCTL_IS_PATHNAME_VALID";
74686|         case FSCTL_MARK_VOLUME_DIRTY : return
| "FSCTL_MARK_VOLUME_DIRTY";
74687|         case FSCTL_QUERY_RETRIEVAL_POINTERS
| : return "FSCTL_QUERY_RETRIEVAL_POINTERS";
74688|         case FSCTL_GET_COMPRESSION : return
| "FSCTL_GET_COMPRESSION";
74689|         case FSCTL_SET_COMPRESSION : return
| "FSCTL_SET_COMPRESSION";
74690|         case FSCTL_MARK_AS_SYSTEM_HIVE : return
| "FSCTL_MARK_AS_SYSTEM_HIVE";
74691|         case FSCTL_OPLOCK_BREAK_ACK_NO_2 :
| return "FSCTL_OPLOCK_BREAK_ACK_NO_2";
74692|         case FSCTL_INVALIDATE_VOLUMES : return
| "FSCTL_INVALIDATE_VOLUMES";
74693|         case FSCTL_QUERY_FAT_BPB : return
| "FSCTL_QUERY_FAT_BPB";
74694|         case FSCTL_REQUEST_FILTER_OPLOCK :
| return "FSCTL_REQUEST_FILTER_OPLOCK";
74695|         case FSCTL_FILESYSTEM_GET_STATISTICS :
| return "FSCTL_FILESYSTEM_GET_STATISTICS";
74696|         #if(_WIN32_WINNT >= 0x0400)
74697|         case FSCTL_GET_NTFS_VOLUME_DATA :
| return "FSCTL_GET_NTFS_VOLUME_DATA";
74698|         case FSCTL_GET_NTFS_FILE_RECORD :
| return "FSCTL_GET_NTFS_FILE_RECORD";
74699|         case FSCTL_GET_VOLUME_BITMAP : return
| "FSCTL_GET_VOLUME_BITMAP";
74700|         case FSCTL_GET_RETRIEVAL_POINTERS :
| return "FSCTL_GET_RETRIEVAL_POINTERS";
74701|         case FSCTL_MOVE_FILE : return
| "FSCTL_MOVE_FILE";
74702|         case FSCTL_IS_VOLUME_DIRTY : return
| "FSCTL_IS_VOLUME_DIRTY";
74703|         case FSCTL_GET_HFS_INFORMATION : return
| "FSCTL_GET_HFS_INFORMATION";
74704|         case FSCTL_ALLOW_EXTENDED_DASD_IO :
| return "FSCTL_ALLOW_EXTENDED_DASD_IO";
74705|         #endif /* _WIN32_WINNT >= 0x0400 */
74706|
74707|         #if(_WIN32_WINNT >= 0x0500)
74708|         case FSCTL_READ_PROPERTY_DATA : return
| "FSCTL_READ_PROPERTY_DATA";
74709|         case FSCTL_WRITE_PROPERTY_DATA : return
| "FSCTL_WRITE_PROPERTY_DATA";

```



```

74710|         case FSCTL_FIND_FILES_BY_SID : return
| "FSCTL_FIND_FILES_BY_SID";
74711|         case FSCTL_DUMP_PROPERTY_DATA : return
| "FSCTL_DUMP_PROPERTY_DATA";
74712|         case FSCTL_SET_OBJECT_ID : return
| "FSCTL_SET_OBJECT_ID";
74713|         case FSCTL_GET_OBJECT_ID : return
| "FSCTL_GET_OBJECT_ID";
74714|         case FSCTL_DELETE_OBJECT_ID :
| return "FSCTL_DELETE_OBJECT_ID";
74715|         case FSCTL_SET_REPARSE_POINT : return
| "FSCTL_SET_REPARSE_POINT";
74716|         case FSCTL_GET_REPARSE_POINT : return
| "FSCTL_GET_REPARSE_POINT";
74717|         case FSCTL_DELETE_REPARSE_POINT :
| return "FSCTL_DELETE_REPARSE_POINT";
74718|         case FSCTL_ENUM_USN_DATA : return
| "FSCTL_ENUM_USN_DATA";
74719|         case FSCTL_SECURITY_ID_CHECK : return
| "FSCTL_SECURITY_ID_CHECK";
74720|         case FSCTL_READ_USN_JOURNAL :
| return "FSCTL_READ_USN_JOURNAL";
74721|         case FSCTL_SET_OBJECT_ID_EXTENDED :
| return "FSCTL_SET_OBJECT_ID_EXTENDED";
74722|         case FSCTL_CREATE_OR_GET_OBJECT_ID :
| return "FSCTL_CREATE_OR_GET_OBJECT_ID";
74723|         case FSCTL_SET_SPARSE : return
| "FSCTL_SET_SPARSE";
74724|         case FSCTL_SET_ZERO_DATA : return
| "FSCTL_SET_ZERO_DATA";
74725|         case FSCTL_QUERY_ALLOCATED_RANGES :
| return "FSCTL_QUERY_ALLOCATED_RANGES";
74726|         case FSCTL_ENABLE_UPGRADE : return
| "FSCTL_ENABLE_UPGRADE";
74727|         case FSCTL_SET_ENCRYPTION : return
| "FSCTL_SET_ENCRYPTION";
74728|         case FSCTL_ENCRYPTION_FSCTL_IO : return
| "FSCTL_ENCRYPTION_FSCTL_IO";
74729|         case FSCTL_WRITE_RAW_ENCRYPTED : return
| "FSCTL_WRITE_RAW_ENCRYPTED";
74730|         case FSCTL_READ_RAW_ENCRYPTED : return
| "FSCTL_READ_RAW_ENCRYPTED";
74731|         case FSCTL_CREATE_USN_JOURNAL : return
| "FSCTL_CREATE_USN_JOURNAL";
74732|         case FSCTL_READ_FILE_USN_DATA : return
| "FSCTL_READ_FILE_USN_DATA";
74733|         case FSCTL_WRITE_USN_CLOSE_RECORD :
| return "FSCTL_WRITE_USN_CLOSE_RECORD";
74734|         case FSCTL_EXTEND_VOLUME : return
| "FSCTL_EXTEND_VOLUME";

```

```

74735|         case FSCTL_QUERY_USN_JOURNAL : return
74736|         | "FSCTL_QUERY_USN_JOURNAL";
74737|         case FSCTL_DELETE_USN_JOURNAL : return
74738|         | "FSCTL_DELETE_USN_JOURNAL";
74739|         case FSCTL_MARK_HANDLE : return
74740|         | "FSCTL_MARK_HANDLE";
74741|         case FSCTL_SIS_COPYFILE : return
74742|         | "FSCTL_SIS_COPYFILE";
74743|         case FSCTL_SIS_LINK_FILES : return
74744|         | "FSCTL_SIS_LINK_FILES";
74745|         case FSCTL_HSM_MSG : return
74746|         | "FSCTL_HSM_MSG";
74747|         case FSCTL_NSS_CONTROL : return
74748|         | "FSCTL_NSS_CONTROL";
74749|         case FSCTL_HSM_DATA : return
74750|         | "FSCTL_HSM_DATA";
74751|         case FSCTL_RECALL_FILE : return
74752|         | "FSCTL_RECALL_FILE";
74753|         case FSCTL_NSS_RCONTROL : return
74754|         | "FSCTL_NSS_RCONTROL";
74755|         #endif /* _WIN32_WINNT >= 0x0500 */
74756|         default:
74757|             return "FSCTL_(Unknown User Request)";
74758|     }
74759| }
74760| case IRP_MN_MOUNT_VOLUME : return
74761| | "FSCTL_(MOUNT_VOLUME)";
74762| case IRP_MN_VERIFY_VOLUME : return
74763| | "FSCTL_(VERIFY_VOLUME)";
74764| case IRP_MN_LOAD_FILE_SYSTEM : return
74765| | "FSCTL_(LOAD_FILE_SYSTEM)";
74766| case IRP_MN_KERNEL_CALL : return
74767| | "FSCTL_(KERNEL_CALL)";
74768| default:
74769|     return "FSCTL_(Unknown Minor function)";
74770| }
74771| return "FSCTL_(Unknown, shouldnt ever get this)";
74772| }
74773|
74774|
74775|
74776| #endif
74777|
74778|
74779|
74780| File Listing: sfilter.h
74781|
74782| //
74783| // Define the local routines used by this driver
74784| | module. This includes a

```

```
74770| // a sample of how to filter a create file operation,
    | and then invoke an I/O
74771| // completion routine when the file has successfully
    | been created/opened.
74772| //
74773|
74774| NTSTATUS
74775| FilterDriverEntry(
74776|     IN PDRIVER_OBJECT DriverObject,
74777|     IN PUNICODE_STRING RegistryPath
74778| );
74779|
74780|
74781| NTSTATUS
74782| SfCreate(
74783|     IN PDEVICE_OBJECT DeviceObject,
74784|     IN PIRP Irp
74785| );
74786|
74787| NTSTATUS
74788| PSManFSPassThru(
74789|     IN PDEVICE_OBJECT DeviceObject,
74790|     IN PIRP Irp
74791| );
74792|
74793|
74794| STATIC
74795| NTSTATUS
74796| SfCreateCompletion(
74797|     IN PDEVICE_OBJECT DeviceObject,
74798|     IN PIRP Irp,
74799|     IN PVOID Context
74800| );
74801|
74802| STATIC
74803| NTSTATUS
74804| SfFsControl(
74805|     IN PDEVICE_OBJECT DeviceObject,
74806|     IN PIRP Irp
74807| );
74808|
74809| STATIC
74810| VOID
74811| SfFsNotification(
74812|     IN PDEVICE_OBJECT DeviceObject,
74813|     IN BOOLEAN FsActive
74814| );
74815|
74816| STATIC
74817| BOOLEAN
```

```

74818| SfFastIoCheckIfPossible(
74819|     IN PFILE_OBJECT FileObject,
74820|     IN PLARGE_INTEGER FileOffset,
74821|     IN ULONG Length,
74822|     IN BOOLEAN Wait,
74823|     IN ULONG LockKey,
74824|     IN BOOLEAN CheckForReadOperation,
74825|     OUT PIO_STATUS_BLOCK IoStatus,
74826|     IN PDEVICE_OBJECT DeviceObject
74827| );
74828|
74829| STATIC
74830| BOOLEAN
74831| SfFastIoRead(
74832|     IN PFILE_OBJECT FileObject,
74833|     IN PLARGE_INTEGER FileOffset,
74834|     IN ULONG Length,
74835|     IN BOOLEAN Wait,
74836|     IN ULONG LockKey,
74837|     OUT PVOID Buffer,
74838|     OUT PIO_STATUS_BLOCK IoStatus,
74839|     IN PDEVICE_OBJECT DeviceObject
74840| );
74841|
74842| STATIC
74843| BOOLEAN
74844| SfFastIoWrite(
74845|     IN PFILE_OBJECT FileObject,
74846|     IN PLARGE_INTEGER FileOffset,
74847|     IN ULONG Length,
74848|     IN BOOLEAN Wait,
74849|     IN ULONG LockKey,
74850|     IN PVOID Buffer,
74851|     OUT PIO_STATUS_BLOCK IoStatus,
74852|     IN PDEVICE_OBJECT DeviceObject
74853| );
74854|
74855| ULONG EnableWritesToNewFiles();
74856| ULONG DisableWritesToNewFiles();
74857|
74858|
74859| NTSTATUS
74860| SfFsWrite(
74861|     PDEVICE_OBJECT DeviceObject,
74862|     PIRP Irp
74863| );
74864| NTSTATUS
74865| PSMANFSClose(
74866|     IN PDEVICE_OBJECT DeviceObject,
74867|     IN PIRP Irp

```

```

74868| );
74869|
74870|
74871| STATIC
74872| BOOLEAN
74873| SfFastIoQueryBasicInfo(
74874|     IN PFILE_OBJECT FileObject,
74875|     IN BOOLEAN Wait,
74876|     OUT PFILE_BASIC_INFORMATION Buffer,
74877|     OUT PIO_STATUS_BLOCK IoStatus,
74878|     IN PDEVICE_OBJECT DeviceObject
74879| );
74880|
74881| STATIC
74882| BOOLEAN
74883| SfFastIoQueryStandardInfo(
74884|     IN PFILE_OBJECT FileObject,
74885|     IN BOOLEAN Wait,
74886|     OUT PFILE_STANDARD_INFORMATION Buffer,
74887|     OUT PIO_STATUS_BLOCK IoStatus,
74888|     IN PDEVICE_OBJECT DeviceObject
74889| );
74890|
74891| STATIC
74892| BOOLEAN
74893| SfFastIoLock(
74894|     IN PFILE_OBJECT FileObject,
74895|     IN PLARGE_INTEGER FileOffset,
74896|     IN PLARGE_INTEGER Length,
74897|     PEPROCESS ProcessId,
74898|     ULONG Key,
74899|     BOOLEAN FailImmediately,
74900|     BOOLEAN ExclusiveLock,
74901|     OUT PIO_STATUS_BLOCK IoStatus,
74902|     IN PDEVICE_OBJECT DeviceObject
74903| );
74904|
74905| STATIC
74906| BOOLEAN
74907| SfFastIoUnlockSingle(
74908|     IN PFILE_OBJECT FileObject,
74909|     IN PLARGE_INTEGER FileOffset,
74910|     IN PLARGE_INTEGER Length,
74911|     PEPROCESS ProcessId,
74912|     ULONG Key,
74913|     OUT PIO_STATUS_BLOCK IoStatus,
74914|     IN PDEVICE_OBJECT DeviceObject
74915| );
74916|
74917| STATIC

```

```

74918| BOOLEAN
74919| SfFastIoUnlockAll(
74920|     IN PFILE_OBJECT FileObject,
74921|     PEPROCESS ProcessId,
74922|     OUT PIO_STATUS_BLOCK IoStatus,
74923|     IN PDEVICE_OBJECT DeviceObject
74924| );
74925|
74926| STATIC
74927| BOOLEAN
74928| SfFastIoUnlockAllByKey(
74929|     IN PFILE_OBJECT FileObject,
74930|     PVOID ProcessId,
74931|     ULONG Key,
74932|     OUT PIO_STATUS_BLOCK IoStatus,
74933|     IN PDEVICE_OBJECT DeviceObject
74934| );
74935|
74936| STATIC
74937| BOOLEAN
74938| SfFastIoDeviceControl(
74939|     IN PFILE_OBJECT FileObject,
74940|     IN BOOLEAN Wait,
74941|     IN PVOID InputBuffer OPTIONAL,
74942|     IN ULONG InputBufferLength,
74943|     OUT PVOID OutputBuffer OPTIONAL,
74944|     IN ULONG OutputBufferLength,
74945|     IN ULONG IoControlCode,
74946|     OUT PIO_STATUS_BLOCK IoStatus,
74947|     IN PDEVICE_OBJECT DeviceObject
74948| );
74949|
74950| STATIC
74951| VOID
74952| SfFastIoDetachDevice(
74953|     IN PDEVICE_OBJECT SourceDevice,
74954|     IN PDEVICE_OBJECT TargetDevice
74955| );
74956|
74957| STATIC
74958| BOOLEAN
74959| SfFastIoQueryNetworkOpenInfo(
74960|     IN PFILE_OBJECT FileObject,
74961|     IN BOOLEAN Wait,
74962|     OUT PFILE_NETWORK_OPEN_INFORMATION Buffer,
74963|     OUT PIO_STATUS_BLOCK IoStatus,
74964|     IN PDEVICE_OBJECT DeviceObject
74965| );
74966|
74967| STATIC

```

```
74968| BOOLEAN
74969| SfFastIoMdlRead(
74970|     IN PFILE_OBJECT FileObject,
74971|     IN PLARGE_INTEGER FileOffset,
74972|     IN ULONG Length,
74973|     IN ULONG LockKey,
74974|     OUT PMDL *MdlChain,
74975|     OUT PIO_STATUS_BLOCK IoStatus,
74976|     IN PDEVICE_OBJECT DeviceObject
74977| );
74978|
74979|
74980| STATIC
74981| BOOLEAN
74982| SfFastIoMdlReadComplete(
74983|     IN PFILE_OBJECT FileObject,
74984|     IN PMDL MdlChain,
74985|     IN PDEVICE_OBJECT DeviceObject
74986| );
74987|
74988| STATIC
74989| BOOLEAN
74990| SfFastIoPrepareMdlWrite(
74991|     IN PFILE_OBJECT FileObject,
74992|     IN PLARGE_INTEGER FileOffset,
74993|     IN ULONG Length,
74994|     IN ULONG LockKey,
74995|     OUT PMDL *MdlChain,
74996|     OUT PIO_STATUS_BLOCK IoStatus,
74997|     IN PDEVICE_OBJECT DeviceObject
74998| );
74999|
75000| STATIC
75001| BOOLEAN
75002| SfFastIoMdlWriteComplete(
75003|     IN PFILE_OBJECT FileObject,
75004|     IN PLARGE_INTEGER FileOffset,
75005|     IN PMDL MdlChain,
75006|     IN PDEVICE_OBJECT DeviceObject
75007| );
75008|
75009| STATIC
75010| BOOLEAN
75011| SfFastIoReadCompressed(
75012|     IN PFILE_OBJECT FileObject,
75013|     IN PLARGE_INTEGER FileOffset,
75014|     IN ULONG Length,
75015|     IN ULONG LockKey,
75016|     OUT PVOID Buffer,
75017|     OUT PMDL *MdlChain,
```

```

75018| OUT PIO_STATUS_BLOCK IoStatus,
75019| OUT struct _COMPRESSED_DATA_INFO
75020| | *CompressedDataInfo,
75021| IN ULONG CompressedDataInfoLength,
75022| IN PDEVICE_OBJECT DeviceObject
75023| );
75024|
75025| STATIC
75026| BOOLEAN
75027| SfFastIoWriteCompressed(
75028| IN PFILE_OBJECT FileObject,
75029| IN PLARGE_INTEGER FileOffset,
75030| IN ULONG Length,
75031| IN ULONG LockKey,
75032| IN PVOID Buffer,
75033| OUT PMDL *MdlChain,
75034| OUT PIO_STATUS_BLOCK IoStatus,
75035| IN struct _COMPRESSED_DATA_INFO
75036| | *CompressedDataInfo,
75037| IN ULONG CompressedDataInfoLength,
75038| IN PDEVICE_OBJECT DeviceObject
75039| );
75040|
75041| STATIC
75042| BOOLEAN
75043| SfFastIoMdlReadCompleteCompressed(
75044| IN PFILE_OBJECT FileObject,
75045| IN PMDL MdlChain,
75046| IN PDEVICE_OBJECT DeviceObject
75047| );
75048|
75049| STATIC
75050| BOOLEAN
75051| SfFastIoMdlWriteCompleteCompressed(
75052| IN PFILE_OBJECT FileObject,
75053| IN PLARGE_INTEGER FileOffset,
75054| IN PMDL MdlChain,
75055| IN PDEVICE_OBJECT DeviceObject
75056| );
75057|
75058| STATIC
75059| BOOLEAN
75060| SfFastIoQueryOpen(
75061| IN PIRP Irp,
75062| OUT PFILE_NETWORK_OPEN_INFORMATION
75063| | NetworkInformation,
75064| IN PDEVICE_OBJECT DeviceObject
75065| );
75066|
75067| STATIC

```



```
75065| NTSTATUS
75066| SfMountCompletion(
75067|     IN PDEVICE_OBJECT DeviceObject,
75068|     IN PIRP Irp,
75069|     IN PVOID Context
75070| );
75071|
75072| STATIC
75073| NTSTATUS
75074| SfLoadFsCompletion(
75075|     IN PDEVICE_OBJECT DeviceObject,
75076|     IN PIRP Irp,
75077|     IN PVOID Context
75078| );
75079|
75080| STATIC
75081| void
75082| SfFastIoReleaseFileForNtCreateSection(
75083|     IN PFILE_OBJECT FileObject
75084| );
75085| STATIC
75086| void
75087| SfFastIoAcquireFileForNtCreateSection(
75088|     IN PFILE_OBJECT FileObject
75089| );
75090| STATIC
75091| NTSTATUS
75092| SfFastIoReleaseForCcFlush(
75093|     IN PFILE_OBJECT FileObject,
75094|     IN PDEVICE_OBJECT DeviceObject
75095| );
75096| STATIC
75097| NTSTATUS
75098| SfFastIoAcquireForCcFlush(
75099|     IN PFILE_OBJECT FileObject,
75100|     IN PDEVICE_OBJECT DeviceObject
75101| );
75102| STATIC
75103| NTSTATUS
75104| SfFastIoReleaseForModWrite(
75105|     IN PFILE_OBJECT FileObject,
75106|     IN PERESOURCE ResourceToRelease,
75107|     IN PDEVICE_OBJECT DeviceObject
75108| );
75109| STATIC
75110| NTSTATUS
75111| SfFastIoAcquireForModWrite(
75112|     IN PFILE_OBJECT FileObject,
75113|     IN PLARGE_INTEGER EndingOffset,
75114|     OUT PERESOURCE *ResourceToRelease,
```

```

75115| IN PDEVICE_OBJECT DeviceObject
75116| );
75117|
75118| NTSTATUS
75119| SfFsDirectoryControl(
75120| IN PDEVICE_OBJECT DeviceObject,
75121| IN PIRP Irp
75122| );
75123| NTSTATUS
75124| SfFsQueryInformation(
75125| IN PDEVICE_OBJECT DeviceObject,
75126| IN PIRP Irp
75127| );
75128| NTSTATUS
75129| SfFsSetInformation(
75130| IN PDEVICE_OBJECT DeviceObject,
75131| IN PIRP Irp
75132| );
75133|
75134|
75135|
75136|
75137| // from ntifs.h
75138|
75139|
75140| //++
75141| //
75142| // VOID
75143| // FsRtlEnterFileSystem (
75144| // );
75145| //
75146| // Routine Description:
75147| //
75148| // This routine is used when entering a file
75149| // | system (e.g., through its
75150| // | system cannot be suspended
75151| // | while running and thus block other file I/O
75152| // | requests. Upon exit
75153| // | the file system must call FsRtlExitFileSystem.
75154| //
75155| // Arguments:
75156| //
75157| // Return Value:
75158| //
75159| // None.
75160| //
75161| //--
75162|
75163| #define FsRtlEnterFileSystem() { \

```

```

75162| KeEnterCriticalRegion(); \
75163| }
75164|
75165| //++
75166| //
75167| // VOID
75168| // FsRtlExitFileSystem (
75169| // );
75170| //
75171| // Routine Description:
75172| //
75173| // This routine is used when exiting a file system
    | (e.g., through its
75174| // Fsd entry point).
75175| //
75176| // Arguments:
75177| //
75178| // Return Value:
75179| //
75180| // None.
75181| //
75182| //--
75183|
75184| #define FsRtlExitFileSystem() { \
75185| KeLeaveCriticalRegion(); \
75186| }
75187|
75188| //
75189| // Define driver FS notification change routine type.
75190| //
75191|
75192| typedef
75193| VOID
75194| (*PDRIVER_FS_NOTIFICATION) (
75195| IN struct _DEVICE_OBJECT *DeviceObject,
75196| IN BOOLEAN FsActive
75197| );
75198|
75199|
75200| NTKERNELAPI
75201| VOID
75202| IoRegisterFileSystem(
75203| IN OUT PDEVICE_OBJECT DeviceObject
75204| );
75205|
75206| NTKERNELAPI
75207| NTSTATUS
75208| IoRegisterFsRegistrationChange(
75209| IN PDRIVER_OBJECT DriverObject,
75210| IN PDRIVER_FS_NOTIFICATION

```

```

    | DriverNotificationRoutine
75211|    );
75212|
75213| typedef struct _OPEN_SNAPSHOT_FILES {
75214|     LIST_ENTRY    ListEntry;
75215|     PFILE_OBJECT  FileObject;
75216|     ULONG         ReadOnly:1;
75217| } tOpenSnapShotFiles, *pOpenSnapShotFiles;
75218|
75219| #define FILE_SYSTEM_UNKNOWN 0
75220| #define FILE_SYSTEM_NTFS    1
75221| #define FILE_SYSTEM_FAT     2
75222|
75223|
75224|
75225| File Listing: SHUTDOWN.cpp
75226|
75227| #include "precomp.h"
75228|
75229|
75230| /*-----
    | -----*/
75231| NTSTATUS
75232| PSMANShutdown(
75233|     IN PDEVICE_OBJECT DeviceObject,
75234|     IN PIRP Irp
75235| )
75236|
75237| /*++
75238|
75239| Routine Description:
75240|
75241|     Pass irp to handler
75242|
75243| Arguments:
75244|
75245|     DriverObject - Pointer to device object to being
    | shutdown by system.
75246|     Irp          - IRP involved.
75247|
75248| Return Value:
75249|
75250|     NT Status
75251|
75252| --*/
75253|
75254| {
75255|     NTSTATUS Status;
75256|
75257|     Debug(DEBUG_PROCCALL,("PSMANShutdown Called\n"));

```

```

75258| switch(PsmGetObjectType(DeviceObject)) {
75259|     case OBJECT_INTERNAL :
75260|         Status = PSMANShutdownObject(DeviceObject,
75261| | Irp);
75262|         break;
75263|     case OBJECT_FILTEREDDISK :
75264|         Status = PSMANShutdownDevice(DeviceObject,
75265| | Irp);
75266|         break;
75267|     case OBJECT_VIRTUALDISK :
75268|         Status = PSMANShutdownVDisk(DeviceObject,
75269| | Irp);
75270|         break;
75271|     case OBJECT_FS_FILTER :
75272|         Status =
75273| | PSMANShutdownFSFilter(DeviceObject, Irp);
75274|         break;
75275|     case OBJECT_FS_OBJECT :
75276|         Status =
75277| | PSMANShutdownFSObject(DeviceObject, Irp);
75278|         break;
75279|     default:
75280|         Irp->IoStatus.Status = Status =
75281| | STATUS_NO_SUCH_DEVICE;
75282|         Irp->IoStatus.Information = 0 ;
75283|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
75284|         break;
75285| }
75286| Debug(DEBUG_PROCCALL,("PSMANShutdown Done\n"));
75287| return Status;
75288| } // end PSMANShutdown()
75289|
75290|
75291|
75292| STATIC NTSTATUS
75293| PSMANShutdownObject(
75294|     IN PDEVICE_OBJECT DeviceObject,
75295|     IN PIRP Irp
75296| )
75297| /*++
75298|
75299| Routine Description:
75300|
75301| This routine is called for a shutdown. These are
75302| | sent by the
75303| system before it actually shuts down.
75304|
75305| Arguments:
75306|

```

```

75301|   DriverObject - Pointer to device object to being
      | shutdown by system.
75302|   Irp          - IRP involved.
75303|
75304| Return Value:
75305|
75306|   NT Status
75307|
75308| --*/
75309|
75310| {
75311|   NOT_REFERENCED(DeviceObject);
75312|   Debug(DEBUG_PROCCALL,("PSManShutdownObject
      | Called\n"));
75313|   Irp->IoStatus.Status = STATUS_SUCCESS;
75314|   Irp->IoStatus.Information = 0;
75315|
75316|   // disable all virtual volumes so no io occurs to
      | them as this device is being shutdown
75317|   GetSnapshotForRead();
75318|   __try {
75319|       // dismount all volumes
75320|       PDEVICE_OBJECT DevObj =
      | PSMAN_DRIVER_OBJECT->DeviceObject;
75321|
75322|       while(DevObj) {
75323|           if (
      | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
75324|
      | Rebuild_DismountAllVolumes(DevObj,TRUE);
75325|           }
75326|           DevObj=DevObj->NextDevice;
75327|       } // while
75328|   } __except(
      | ExceptionFilter(GetExceptionInformation()) ) {
75329|       Debug(DEBUG_SFILTER,("Exception %08x in
      | ShutdownObject\n",GetExceptionCode()));
75330|   }
75331|
75332|   ReleaseSnapshotForRead();
75333|
75334|
75335|   // shutdown all io processing from this point on.
75336|   // we should probabally find a better way to do it
      | incase
75337|   // the io after we shutdown is important.
75338|   // This is done to solve a deadlock when io
75339|   // comes down after we got a shutdown command from
      | the os
75340|   // for our object.

```

```

75341|   GlobalData->ShutDownCalled=TRUE;
75342|
75343|   IoCompleteRequest(Irp, IO_NO_INCREMENT);
75344|   Debug(DEBUG_PROCCALL,("PSManShutdownObject
    | Done\n"));
75345|   return STATUS_SUCCESS;
75346|
75347| } // end PSManShutdownObject()
75348|
75349|
75350| /*-----
    | -----*/
75351| STATIC NTSTATUS
75352| PSManShutdownDevice(
75353|   IN PDEVICE_OBJECT DeviceObject,
75354|   IN PIRP Irp
75355|   )
75356|
75357| /*++
75358|
75359| Routine Description:
75360|
75361|   This routine is called for a shutdown. These are
    | sent by the
75362|   system before it actually shuts down.
75363|
75364| Arguments:
75365|
75366|   DriverObject - Pointer to device object to being
    | shutdown by system.
75367|   Irp          - IRP involved.
75368|
75369| Return Value:
75370|
75371|   NT Status
75372|
75373| --*/
75374|
75375| {
75376|   NTSTATUS Status;
75377|   PIO_STACK_LOCATION currentIrpStack =
    | IoGetCurrentIrpStackLocation(Irp);
75378|
75379|   TRACE( TRACE_SHUTDOWN,
75380|
    | currentIrpStack->Parameters.Read.ByteOffset.HighPart,
75381|
    | currentIrpStack->Parameters.Read.ByteOffset.LowPart,
75382|   currentIrpStack->Parameters.Read.Length,
75383|   currentIrpStack->Parameters.Read.Key,

```

```

75384|         "");
75385|     Debug(DEBUG_SHUTDOWN |
        | DEBUG_PROCCALL,("PSManShutdownDevice Called Device=%p,
        | Irp=%p\n",DeviceObject,Irp));
75386|
75387|     Status = PSManPassThru( DeviceObject, Irp );
75388|     Debug(DEBUG_SHUTDOWN |
        | DEBUG_PROCCALL,("PSManShutdownDevice Done Device=%p,
        | Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
75389|     return Status;
75390| } // end PSManShutdownDevice()
75391|
75392| /*-----
        | -----*/
75393| STATIC NTSTATUS PSManShutdownVDisk(
75394|     IN PDEVICE_OBJECT DeviceObject,
75395|     IN PIRP Irp
75396| )
75397| {
75398|     NTSTATUS Status=STATUS_SUCCESS;
75399|
75400|     Debug(DEBUG_PROCCALL |
        | DEBUG_SHUTDOWN,("PSManShutdownVDisk Called Dev=%p,
        | Irp=%p\n",DeviceObject,Irp));
75401|     Irp->IoStatus.Information = 0;
75402|     Irp->IoStatus.Status = Status;
75403|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
75404|     Debug(DEBUG_PROCCALL |
        | DEBUG_SHUTDOWN,("PSManShutdownVDisk Done\n"));
75405|
75406|     return Status;
75407| }
75408|
75409|
75410| /*-----
        | -----*/
75411| STATIC NTSTATUS
75412| PSManShutdownFSFilter(
75413|     IN PDEVICE_OBJECT DeviceObject,
75414|     IN PIRP Irp
75415| )
75416|
75417| /*++
75418|
75419| Routine Description:
75420|
75421|     This routine is called for a shutdown. These are
        | sent by the
75422|     system before it actually shuts down.
75423|

```



```

75424| Arguments:
75425|
75426|     DriverObject - Pointer to device object to being
        | shutdown by system.
75427|     Irp         - IRP involved.
75428|
75429| Return Value:
75430|
75431|     NT Status
75432|
75433| --*/
75434|
75435| {
75436|     NTSTATUS Status;
75437|     PIO_STACK_LOCATION currentIrpStack =
        | IoGetCurrentIrpStackLocation(Irp);
75438|     PFS_FILTER_EXTENSION
        | DevExt=(PFS_FILTER_EXTENSION)GetDeviceExtension(DeviceOb
        | ject);
75439|
75440|
75441|     Debug(DEBUG_SHUTDOWN |
        | DEBUG_PROCCALL,("PSManShutdownFSFilter Called
        | Device=%p, Irp=%p\n",DeviceObject,Irp));
75442|     Debug(DEBUG_SHUTDOWN |
        | DEBUG_PROCCALL,("PSManShutdownFSFilter: Virtual=%08x,
        | PSMStorageObject=%08x\n",DevExt->Virtual,DevExt->PSMStor
        | ageObject));
75443|
75444|     if(!DevExt->Virtual) &&
        | (DevExt->PSMStorageObject)) {
75445|         Debug(DEBUG_SHUTDOWN |
        | DEBUG_PROCCALL,("PSManShutdownFSFilter switching to
        | directio\n"));
75446|         PFILTERED_EXTENSION de = GetFilteredExtension
        | (DevExt->PSMStorageObject);
75447|         de->DoDirectIO = TRUE;
75448|     }
75449|
75450|     Status = PSManFSPassThru( DeviceObject, Irp );
75451|     Debug(DEBUG_SHUTDOWN |
        | DEBUG_PROCCALL,("PSManShutdownFSFilter Done
        | Device=%p, Irp=%p,
        | Status=%08x\n",DeviceObject,Irp,Status));
75452|     return Status;
75453| } // end PSManShutdownFSFilter()
75454|
75455| /*-----
        | -----*/
75456| STATIC NTSTATUS PSManShutdownFSObject(

```

```

75457| IN PDEVICE_OBJECT DeviceObject,
75458| IN PIRP Irp
75459| )
75460| {
75461| NTSTATUS Status=STATUS_SUCCESS;
75462|
75463| Debug(DEBUG_PROCCALL |
    | DEBUG_SHUTDOWN,("PSManShutdownFSObject Called Dev=%p,
    | Irp=%p\n",DeviceObject,Irp));
75464| Irp->IoStatus.Information = 0;
75465| Irp->IoStatus.Status = Status;
75466| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
75467| Debug(DEBUG_PROCCALL |
    | DEBUG_SHUTDOWN,("PSManShutdownFSObject Done\n"));
75468|
75469| return Status;
75470| }
75471|
75472|
75473|
75474| File Listing: SHUTDOWN.h
75475|
75476| NTSTATUS
75477| PSManShutdown(
75478|     IN PDEVICE_OBJECT DeviceObject,
75479|     IN PIRP Irp
75480| );
75481|
75482| STATIC NTSTATUS
75483| PSManShutdownObject(
75484|     IN PDEVICE_OBJECT DeviceObject,
75485|     IN PIRP Irp
75486| );
75487|
75488| STATIC NTSTATUS
75489| PSManShutdownDevice(
75490|     IN PDEVICE_OBJECT DeviceObject,
75491|     IN PIRP Irp
75492| );
75493| STATIC NTSTATUS PSManShutdownVDisk(
75494|     IN PDEVICE_OBJECT DeviceObject,
75495|     IN PIRP Irp
75496| );
75497| STATIC NTSTATUS PSManShutdownFSObject(
75498|     IN PDEVICE_OBJECT DeviceObject,
75499|     IN PIRP Irp
75500| );
75501| STATIC NTSTATUS PSManShutdownFSFilter(
75502|     IN PDEVICE_OBJECT DeviceObject,
75503|     IN PIRP Irp

```

```

75504| );
75505|
75506|
75507|
75508| File Listing: snapshot.h
75509|
75510| #define _SNAPSHOT_DEFINED_
75511|
75512| // this struct is unique to each snapshot
75513| typedef struct skSnapShotMaster {
75514|     LARGE_INTEGER SnapShotTime;        // time
        | snapshot occurred.
75515|     ULONG Instance;                    // instance
        | number for this snapshot for volume mapping
75516|     ULONG Count;                        // number
        | of snap shots this master has
75517|     HANDLE ExclusiveProcess;           // who has
        | us opened exclusively
75518|     NTSTATUS Status;                   // status
        | that caused this snapshot to be canceled.
75519|     ULONG OutOfSeconds;
75520|     ULONG Persistent;
75521|     PVOID DllPrivateUse;
75522|     ULONG GroupNumber;
75523|     ULONG NumToKeep;
75524|     unsigned char Priority;
75525|     unsigned char SnapShotFlags;
75526|     unsigned char Reserved1;
75527|     unsigned char Reserved2;
75528|
75529|     WCHAR UserSnapShotName[256];
75530|
75531|     LIST_ENTRY SnapShots;
75532| } tkSnapShotMaster,*pkSnapShotMaster;
75533|
75534| // this struct is unique to each device
75535| typedef struct skSnapShotEntry {
75536|     pkSnapShotMaster MasterSnapShot;
75537|
75538| // specific to kernel only
75539|     PDEVICE_OBJECT DeviceObject; // device this
        | snapshot belongs to
75540|     ULONG Count;                    // how many
        | people have this snapshot in use
75541|     BOOLEAN Deleted;                // whether a
        | delete is pending
75542|     PRTL_BITMAP PSM Sectors; // what sectors to
        | PSM and not to PSM.
75543|     pDictionary Dictionary;
75544|     LIST_ENTRY DevExt;              // Linked list for

```

```

| Device Objects
75545| LIST_ENTRY    Master;    // Linked list for
| Master
75546| LIST_ENTRY    User;      // Linked list for
| User Snapshots
75547| } tkSnapShotEntry,*pkSnapShotEntry;
75548|
75549|
75550|
75551| File Listing: SP4.h
75552|
75553| #undef IsRecognizedPartition
75554| #undef IsContainerPartition
75555|
75556| // begin_winioctl
75557| //
75558| // Define the partition types returnable by known disk
| drivers.
75559| //
75560|
75561| #define PARTITION_ENTRY_UNUSED    0x00    //
| Entry unused
75562| #define PARTITION_FAT_12          0x01    //
| 12-bit FAT entries
75563| #define PARTITION_XENIX_1        0x02    //
| Xenix
75564| #define PARTITION_XENIX_2        0x03    //
| Xenix
75565| #define PARTITION_FAT_16          0x04    //
| 16-bit FAT entries
75566| #define PARTITION_EXTENDED        0x05    //
| Extended partition entry
75567| #define PARTITION_HUGE            0x06    //
| Huge partition MS-DOS V4
75568| #define PARTITION_IFS            0x07    //
| IFS Partition
75569| #define PARTITION_FAT32          0x0B    //
| FAT32
75570| #define PARTITION_FAT32_XINT13    0x0C    //
| FAT32 using extended int13 services
75571| #define PARTITION_XINT13          0x0E    //
| Win95 partition using extended int13 services
75572| #define PARTITION_XINT13_EXTENDED 0x0F    //
| Same as type 5 but uses extended int13 services
75573| #define PARTITION_PREP            0x41    //
| PowerPC Reference Platform (PReP) Boot Partition
75574| #define PARTITION_LDM            0x42    //
| Logical Disk Manager partition
75575| #define PARTITION_UNIX            0x63    //
| Unix

```

```

75576|
75577| #define VALID_NTFT          0xC0    //
    | NTFT uses high order bits
75578|
75579| //
75580| // The high bit of the partition type code indicates
    | that a partition
75581| // is part of an NTFT mirror or striped array.
75582| //
75583|
75584| #define PARTITION_NTFT        0x80    //
    | NTFT partition
75585|
75586| //
75587| // The following macro is used to determine which
    | partitions should be
75588| // assigned drive letters.
75589| //
75590|
75591| //++
75592| //
75593| // BOOLEAN
75594| // IsRecognizedPartition(
75595| //     IN ULONG PartitionType
75596| // )
75597| //
75598| // Routine Description:
75599| //
75600| //     This macro is used to determine to which
    | partitions drive letters
75601| //     should be assigned.
75602| //
75603| // Arguments:
75604| //
75605| //     PartitionType - Supplies the type of the
    | partition being examined.
75606| //
75607| // Return Value:
75608| //
75609| //     The return value is TRUE if the partition type
    | is recognized,
75610| //     otherwise FALSE is returned.
75611| //
75612| //--
75613|
75614| #define IsRecognizedPartition( PartitionType ) (
    | \
75615|     ((PartitionType & PARTITION_NTFT) &&
    | ((PartitionType & ~0xC0) == PARTITION_FAT_12)) || \
75616|     ((PartitionType & PARTITION_NTFT) &&

```

```

    | ((PartitionType & ~0xC0) == PARTITION_FAT_16)) || \
75617|    ((PartitionType & PARTITION_NTFT) &&
    | ((PartitionType & ~0xC0) == PARTITION_IFS)) || \
75618|    ((PartitionType & PARTITION_NTFT) &&
    | ((PartitionType & ~0xC0) == PARTITION_HUGE)) || \
75619|    ((PartitionType & PARTITION_NTFT) &&
    | ((PartitionType & ~0xC0) == PARTITION_FAT32)) || \
75620|    ((PartitionType & PARTITION_NTFT) &&
    | ((PartitionType & ~0xC0) == PARTITION_FAT32_XINT13)) ||
    | \
75621|    ((PartitionType & PARTITION_NTFT) &&
    | ((PartitionType & ~0xC0) == PARTITION_XINT13)) || \
75622|    ((PartitionType & ~PARTITION_NTFT) ==
    | PARTITION_FAT_12) || \
75623|    ((PartitionType & ~PARTITION_NTFT) ==
    | PARTITION_FAT_16) || \
75624|    ((PartitionType & ~PARTITION_NTFT) ==
    | PARTITION_IFS) || \
75625|    ((PartitionType & ~PARTITION_NTFT) ==
    | PARTITION_HUGE) || \
75626|    ((PartitionType & ~PARTITION_NTFT) ==
    | PARTITION_FAT32) || \
75627|    ((PartitionType & ~PARTITION_NTFT) ==
    | PARTITION_FAT32_XINT13) || \
75628|    ((PartitionType & ~PARTITION_NTFT) ==
    | PARTITION_XINT13) )
75629|
75630| //++
75631| //
75632| // BOOLEAN
75633| // IsContainerPartition(
75634| //     IN ULONG PartitionType
75635| // )
75636| //
75637| // Routine Description:
75638| //
75639| //     This macro is used to determine to which
    | partition types are actually
75640| //     containers for other partitions (ie, extended
    | partitions).
75641| //
75642| // Arguments:
75643| //
75644| //     PartitionType - Supplies the type of the
    | partition being examined.
75645| //
75646| // Return Value:
75647| //
75648| //     The return value is TRUE if the partition type
    | is a container,

```

```

75649| // otherwise FALSE is returned.
75650| //
75651| //--
75652|
75653| #define IsContainerPartition( PartitionType ) \
75654| ((PartitionType == PARTITION_EXTENDED) ||
75655| | (PartitionType == PARTITION_XINT13_EXTENDED))
75656|
75657|
75658| File Listing: tempdict.cpp
75659|
75660| #include "precomp.h"
75661|
75662| #define INVALID_HANDLE_VALUE ((HANDLE)(-1))
75663|
75664|
75665| TemporaryDictionary::TemporaryDictionary():
75666| PersistentDictionary()
75667| {
75668| Debug(DEBUG_DCPSM,("td::Constructor:
75669| | this=%08x\n",this));
75670| Flags &= ~DICT_FLAG_PERSISTENT;
75671| Flags |= DICT_FLAG_NONPERSISTENT;
75672| }
75673|
75674| TemporaryDictionary::~TemporaryDictionary()
75675| {
75676| Debug(DEBUG_DCPSM,("td::Destructor:
75677| | this=%08x\n",this));
75678| // Don't call pd::cleanup() here, because
75679| | TemporaryDictionary inherits
75680| | from PersistentDictionary, whose destructor will
75681| | call pd::cleanup.
75682| }
75683| /*--- end of file tempdict.cpp ---*/
75684|
75685|
75686|
75687| File Listing: THREAD.cpp
75688|
75689| #include "precomp.h"
75690|
75691| #ifdef ALLOC_PRAGMA_DO_NOT_DO
75692| #pragma alloc_text(PAGE, SbIncrementRunningThreads)
75693| #pragma alloc_text(PAGE, SbDecrementRunningThreads)

```

```

75694| #pragma alloc_text(PAGE, SbThreadQueueUpdate)
75695| #pragma alloc_text(PAGE, SbThreadQueueInc)
75696| #pragma alloc_text(PAGE, SbThreadQueueDec)
75697| #pragma alloc_text(PAGE, SbThreadQueueWait)
75698| #pragma alloc_text(PAGE, SbWaitForThreadWork)
75699| #pragma alloc_text(PAGE, SbWaitOnEvent)
75700| #pragma alloc_text(PAGE, SbThreadInit)
75701| #pragma alloc_text(PAGE, SbGetWorkItem)
75702| #pragma alloc_text(PAGE, SbGetWork)
75703| #pragma alloc_text(PAGE, SbMakeAnotherThread)
75704| #pragma alloc_text(PAGE, SbWaitForWriteFromNT)
75705| #pragma alloc_text(PAGE,
    | SbCompleteNextWriteOnQueue)
75706| #pragma alloc_text(PAGE, SbCompleteWritesOnQueue)
75707| #pragma alloc_text(PAGE, SbWaitForFreeThread)
75708| #pragma alloc_text(PAGE, SbAllocateIrpResources)
75709| #pragma alloc_text(PAGE,
    | SbAllocateIrpResourcesOrWait)
75710| #pragma alloc_text(PAGE, SbInitReadIrp)
75711| #pragma alloc_text(PAGE, SbInitOtherIrpStack)
75712| #pragma alloc_text(PAGE, SbInitReadIrpStack)
75713| #pragma alloc_text(PAGE, SaveOriginalDataThread)
75714| #pragma alloc_text(PAGE, WriteDispatchThread)
75715| #endif
75716|
75717|
75718| NTSTATUS
75719| PSMANSendOrigWrite(
75720|     IN PDEVICE_OBJECT DeviceObject,
75721|     IN PIRP Irp
75722| );
75723|
75724| /*-----
    | -----*/
75725| STATIC void SbIncrementRunningThreads( void )
75726| {
75727|     pmAcquireMutex ( &WorkerThreadMutex, NULL );
75728|     GlobalThreadCount++;
75729|     pmReleaseMutex ( &WorkerThreadMutex );
75730| }
75731|
75732| /*-----
    | -----*/
75733| STATIC void SbDecrementRunningThreads( void )
75734| {
75735|     pmAcquireMutex ( &WorkerThreadMutex, NULL );
75736|     GlobalThreadCount--;
75737|     pmReleaseMutex ( &WorkerThreadMutex );
75738| }
75739|

```



```

75740| #define GetState() KeReadStateEvent(&WorkerThreadEvent)
75741|
75742| /*-----
| -----*/
75743| STATIC int SbThreadQueueUpdate ( int Up )
75744| {
75745|     int Result = TRUE;
75746|     int Signal = 0;
75747|
75748|     TRACE( TRACE_THREADUPDATE, Up, ThreadsAwake,
| NumberOfThreads, 0, "" );
75749|
75750|     pmAcquireMutex ( &WorkerThreadMutex, NULL );
75751|
75752|     //Debug(DEBUG_THREAD,("SbThreadQueueUpdate: Up=%d,
| ThreadsAwake=%d,
| Semacount=%d\n",Up,ThreadsAwake,GetState()));
75753|     if ( Up && (ThreadsAwake < NumberOfThreads) ) {
75754|         ThreadsAwake++;
75755|         //Debug(DEBUG_THREAD,("SbThreadQueueUpdate:
| Going up ThreadsAwake=%d,
| Semacount=%d\n",ThreadsAwake,GetState()));
75756|     } else
75757|         if ( !Up && (ThreadsAwake>0) ) {
75758|             ThreadsAwake--;
75759|             Signal = 1;
75760|             //Debug(DEBUG_THREAD,("SbThreadQueueUpdate:
| Going down ThreadsAwake=%d,
| Semacount=%d\n",ThreadsAwake,GetState()));
75761|         } else {
75762|             Result = FALSE;
75763|             Signal = -1;
75764|             //Debug(DEBUG_THREAD,("SbThreadQueueUpdate:
| Staying sane ThreadsAwake=%d,
| Semacount=%d\n",ThreadsAwake,GetState()));
75765|         }
75766|
75767|     pmReleaseMutex ( &WorkerThreadMutex );
75768|
75769|     // Wake up the thread if he was waiting on us..
75770|     if ( Signal==1 ) {
75771|         pmSetEvent( &WorkerThreadEvent );
75772|     } else
75773|         if ( Signal==-1 ) {
75774|             pmClearEvent(&WorkerThreadEvent);
75775|         }
75776|
75777|     return Result;
75778| }
75779|

```

```

75780| /*-----
| -----*/
75781| STATIC int SbThreadQueueInc ( ) {
75782|     return SbThreadQueueUpdate( 1 );
75783| }
75784|
75785| /*-----
| -----*/
75786| STATIC int SbThreadQueueDec ( ) {
75787|     return SbThreadQueueUpdate( 0 );
75788| }
75789|
75790| // if returns 0 then OK! else error code
75791| // such as not enough resources, or time to exit
75792|
75793| /*-----
| -----*/
75794| STATIC NTSTATUS SbThreadQueueWait ( ) {
75795|     NTSTATUS Status = STATUS_WAIT_0;
75796|     LARGE_INTEGER TimeOut;
75797|     PVOID ObjectTable[2] = { &WorkerThreadEvent,
| &PSManExitingEvent};
75798|
75799|     TimeOut.QuadPart = RELATIVE(MICROSECONDS(((signed
| long)NewThreadStartDelay)));
75800|
75801|     while ( (!SbThreadQueueInc() ) &&
75802|             (Status == STATUS_WAIT_0 )
75803|             ) {
75804|         //Debug(DEBUG_THREAD,("Writer: Out of threads,
| waiting... %d, %d\n",ThreadsAwake,GetState()));
75805|
75806|         ASSERT(KernelIrql() < DISPATCH_LEVEL);
75807|         Status =
| pmWaitForMultipleObjects(ObjectTable,2,&TimeOut);
75808|     }
75809|     return(Status == STATUS_WAIT_0 ? 0 : Status );
75810| }
75811|
75812| /*-----
| -----*/
75813| /*lint -save -e614 */
75814| STATIC NTSTATUS SbWaitOnEvent ( PKEVENT Event ) {
75815|     NTSTATUS Status = STATUS_WAIT_0;
75816|     PVOID ObjectTable[2] = { Event,
| &PSManExitingEvent};
75817|
75818|     ASSERT(KernelIrql() < DISPATCH_LEVEL);
75819|     Status =
| pmWaitForMultipleObjects(ObjectTable,2,NULL);

```

```

75820| // object 1 is exiting event...
75821| // status_wait_1 is a successful status code, so
| return an error..
75822| if ( Status == STATUS_WAIT_1 ) {
75823|     Status = PSM_CANCELED_BY_USER;
75824| }
75825|
75826| return Status;
75827| }
75828| /*lint -restore */
75829|
75830| #ifdef DEBUG
75831| /*-----
| -----*/
75832| STATIC void SbUpdateThreadStatus(
75833|     ULONG
| ThreadNum,
75834|     PDEVICE_OBJECT
| DeviceObject,
75835|     PIRP Irp,
75836|     ULONG State,
75837|     ULARGE_INTEGER Sector,
75838|     ULONG Count,
75839|     ULARGE_INTEGER Current
| )
75840| {
75841|     pmAcquireMutex ( &WorkerThreadMutex, NULL );
75842|
75843|     if ( ThreadObjects ) {
75844|         ThreadObjects[ThreadNum].State = State;
75845|         ThreadObjects[ThreadNum].DeviceObject =
| DeviceObject;
75846|         ThreadObjects[ThreadNum].Irp = Irp;
75847|         ThreadObjects[ThreadNum].Sector = Sector;
75848|         ThreadObjects[ThreadNum].Count = Count;
75849|         ThreadObjects[ThreadNum].Current =
| Current;
75850|     }
75851|     pmReleaseMutex ( &WorkerThreadMutex );
75852| }
75853| /*-----
| -----*/
75854| STATIC void SbUpdateWriterStatus(
75855|     ULONG State,
75856|     ULARGE_INTEGER Sector,
75857|     ULONG Count )
75858| {
75859|     WriteThreadObject.State = State;
75860|     WriteThreadObject.Sector = Sector;
75861|     WriteThreadObject.Count = Count;

```

```

75862| WriteThreadObject.Current.QuadPart = 0;
75863| WriteThreadObject.DeviceObject = NULL;
75864| WriteThreadObject.Irp = NULL;
75865| }
75866| #else
75867| #define
    | SbUpdateThreadStatus(ThreadNum,DeviceObject,Irp,State,Se
    | ctor,Count,Current)
75868| #define SbUpdateWriterStatus(State,Sector,Count)
75869| #endif
75870|
75871| /*-----
    | -----*/
75872| STATIC NTSTATUS SbWaitForThreadWork()
75873| {
75874|     PVOID ObjectTable[2] = { &PSManExitingEvent,
    | &ThreadSemaphore};
75875|
75876|     if ( !SbThreadQueueDec() ) {
75877|         Debug(DEBUG_THREAD,("Error! Thread Count Below
    | 0, ThreadsAwake=%d,
    | Num=%d\n",ThreadsAwake,NumberOfThreads));
75878|         //KeBugCheck ( (ULONG)STATUS_BAD_STACK );
75879|     }
75880|
75881|     ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
75882|     return
    | pmWaitForMultipleObjects(ObjectTable,2,NULL);
75883| }
75884|
75885| STATIC NTSTATUS SbWaitForWriteAfterRead()
75886| {
75887|     PVOID ObjectTable[2] = { &PSManExitingEvent,
    | &WriteAfterReadSemaphore};
75888|
75889|     ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
75890|     return
    | pmWaitForMultipleObjects(ObjectTable,2,NULL);
75891| }
75892|
75893| /*-----
    | -----*/
75894| STATIC NTSTATUS SbThreadInit (
75895|     ULONG ThreadNum
75896| )
75897| {
75898|     NTSTATUS Status=STATUS_SUCCESS;
75899|
75900|     if ( !SbThreadQueueInc() ) {
75901|         Debug(DEBUG_THREAD,("Thread %08x: invalid

```

```

    | Thread Count,ThreadsAwake=%d,
    | Num=%d\n",ThreadNum,ThreadsAwake,NumberOfThreads));
75902|    //KeBugCheck ( (ULONG)STATUS_STACK_OVERFLOW );
75903|    }
75904|
75905|    SbIncrementRunningThreads();
75906|
75907|    // threads normally start with variable priority.
    | This sets it to the
75908|    // lowest of the time critical prioritys
75909|    // Realtime threads have no quantum timeout. They
    | only give up the
75910|    // cpu when they voluntarily go into a wait state,
    | or when preempted by
75911|    // a thread of higher priority
75912|    //KeSetPriorityThread( KeGetCurrentThread(),
    | LOW_REALTIME_PRIORITY );
75913|
75914|    // wait for thread 0 to finish initing
75915|    if ( ThreadNum==0 ) {
75916|        KeSetEvent(&Thread0Inited,(KRIORITY)9,FALSE);
75917|    } else {
75918|        Status = SbWaitOnEvent( &Thread0Inited );
75919|    }
75920|
75921|    return Status;
75922| }
75923|
75924| /*-----
    | -----*/
75925| PIRP SbGetWorkItem ( )
75926| {
75927|     PLIST_ENTRY ListEntry;
75928|
75929|     ListEntry = ExInterlockedRemoveHeadList (
75930|
    | &ThreadsWorkToDoQueue,    // List Head
75931|
    | &ThreadsWorkToDoSpinLock // Lock
75932|
    | );
75933|
75934|     if ( (!ListEntry) ||
    | (ListEntry==&ThreadsWorkToDoQueue) ) {
75935|         Debug(DEBUG_THREAD,("GetWorkItem: ListEntry is
    | empty\n"));
75936|         return NULL;
75937|     }
75938|
75939|     /*lint -save -e413 */
75940|     return(CONTAINING_RECORD( ListEntry, IRP,

```

```

    | Tail.Overlay.ListEntry ));
75941|  /*lint -restore */
75942| }
75943|
75944| /*-----
    | -----*/
75945| STATIC PIRP SbGetWork(
75946|         PDEVICE_OBJECT *DeviceObject,
75947|         PIO_STACK_LOCATION *Stack,
75948|         PFILTERED_EXTENSION *DevExt,
75949|         tWriteRequest **WriteRequest,
75950|         PCHAR *Buffer
75951| )
75952| {
75953|     PIRP Irp;
75954|
75955|     Irp = SbGetWorkItem();
75956|
75957|     if ( Irp ) {
75958|         (*Stack) =
            | IoGetCurrentIrpStackLocation( Irp );
75959|
75960|         (*WriteRequest) =
            | (tWriteRequest*)(*Stack)->Parameters.Others.Argument1);
75961|         (*DeviceObject) =
            | (*WriteRequest)->DeviceObject;
75962|         (*DevExt) =
            | GetFilteredExtension((*DeviceObject));
75963|         (*Buffer) = (PCHAR)
            | (*WriteRequest)->Buffer;
75964|     }
75965|
75966|     return Irp;
75967| }
75968|
75969|
75970|
75971| BOOLEAN DeleteOldestSnapShot( PDEVICE_OBJECT Volume,
    | BOOLEAN DeleteAlwaysKeep )
75972| {
75973|     BOOLEAN SnapShotWasDeleted = FALSE;
75974|     pkSnapShotMaster Oldest=NULL;
75975|     ULONG NumInGroup=0;
75976|     PDEVICE_OBJECT DevObj;
75977|     PFILTERED_EXTENSION DevExt=NULL;
75978|     pkSnapShotEntry p;
75979|
75980|     Debug(DEBUG_DCPSPM,("DeleteOldestSnapShot: volume
    | %08x, Delete Always
    | Keep=%d\n", Volume, DeleteAlwaysKeep));

```

```

75981|
75982|     if(Volume) {
75983|         // work only with this volume
75984|         DevObj = Volume;
75985|     } else {
75986|         // start at the top
75987|         DevObj = PSMAN_DRIVER_OBJECT->DeviceObject;
75988|     }
75989|
75990|     __try {
75991|         // now go through all snapshots for this volume
75992|         | and count
75993|         // the snapshots along with finding the oldest
75994|         while ( DevObj != NULL ) {
75995|             if (
75996|                 | PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK ) {
75997|                 DevExt = GetFilteredExtension(DevObj);
75998|
75999|                 __try {
76000|                     GetSnapShotForRead();
76001|                 } __try {
76002|                     | p=GetTopSnapShot(&DevExt->SnapShots);
76003|                     while ( p ) {
76004|                         if ( (DeleteAlwaysKeep) ||
76005|                             | ( p->MasterSnapShot->Priority!=255 ) ) {
76006|                             if ( Oldest ) {
76007|                                 if (
76008|                                     | p->MasterSnapShot->Priority<Oldest->Priority ) {
76009|                                     // at a lower
76010|                                     | priority so grab it, regardless of time
76011|                                     Oldest =
76012|                                     | p->MasterSnapShot;
76013|                                 } else
76014|                                 if (
76015|                                     | p->MasterSnapShot->Priority==Oldest->Priority ) {
76016|                                     // same
76017|                                     | priority, compare times
76018|                                     if (
76019|                                     | p->MasterSnapShot->SnapShotTime.QuadPart<Oldest->SnapShotTime.QuadPart ) {
76020|                                     // mark
76021|                                     | this one as the oldest and lowest priority found so far
76022|                                     Oldest =
76023|                                     | p->MasterSnapShot;
76024|                                 }
76025|                                 }
76026|                             } else {
76027|                                 // first suitable
76028|                                 | snapshot found

```

```

76017|             Oldest =
76018|         | p->MasterSnapShot;
76019|         }
76020|         } // always keep
76021|         | p=GetNextSnapShot(&DevExt->SnapShots,p);
76022|         }
76023|         } __finally {
76024|             ReleaseSnapShotForRead();
76025|         }
76026|         | __except(ExceptionFilter(GetExceptionInformation())) {
76027|             | Debug(DEBUG_DCPSM,("DeleteOldestSnapShot: Exception
76028|             | %08x for device %08x\n",GetExceptionCode(),DevObj));
76029|         }
76030|         }
76031|         if(Volume) {
76032|             // stop scan, as we only wanted oldest
76033|             | for this volume
76034|             DevObj = NULL;
76035|         } else {
76036|             // continue scan on next volume
76037|             DevObj=DevObj->NextDevice;
76038|         }
76039|         }
76040|         if ( Oldest ) {
76041|             if ( GlobalData->NumActive>1 ) {
76042|                 pOT_USER User =
76043|                 | FindPSMUser(PsGetCurrentProcess(),(_ETHREAD*)-2);
76044|                 | Debug(DEBUG_DCPSM,("DeleteOldestSnapShot: Deleting
76045|                 | Snapshot %08x\n",Oldest));
76046|                 UpdateGlobalStatus
76047|                 | (PSM_DESTROYING_SNAPSHOT);
76048|                 NTSTATUS CloseStatus =
76049|                 | InternalClosePSM(User,Oldest);
76050|                 UpdateGlobalStatus (PSM_IDLE);
76051|                 if ( NT_SUCCESS(CloseStatus) ) {
76052|                     SnapShotWasDeleted = TRUE;
76053|                 }
76054|             } else {
76055|                 | Debug(DEBUG_DCPSM,("DeleteOldestSnapShot: Only 1
76056|                 | snapshot is active!\n"));
76057|             }
76058|         } else {
76059|             Debug(DEBUG_DCPSM,("DeleteOldestSnapShot:
76060|             | Oldest is null, how odd\n"));

```



```

76053| #ifdef DEBUG
76054|         DbgBreakPoint();
76055| #endif
76056|     }
76057| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
76058|     Debug(DEBUG_DCPSM,("DeleteOldestSnapShot:
    | Exception %08x deleting snapshot
    | %08x\n",GetExceptionCode(),Oldest));
76059| }
76060|
76061|     return SnapShotWasDeleted;
76062| }
76063|
76064| static ULONG DeleteCalledCount=0;
76065| LARGE_INTEGER TimeOfLastLogEntry={0};
76066| static ULONG TypeOfLastLogEntry=0;
76067|
76068|
76069| void FindAndDeleteSnapShotsForVolume( PDEVICE_OBJECT
    | Volume, BOOLEAN DeleteAlwaysKeep )
76070| {
76071|     // dont delete last snapshot for volume
76072|     if ( NumberOfSnapShotsForVolume(Volume)>1 ) {
76073|         DeleteOldestSnapShot(Volume,DeleteAlwaysKeep);
76074|     }
76075| }
76076|
76077| void CacheThresholdReached(PDEVICE_OBJECT Volume)
76078| {
76079|     Debug(DEBUG_DCPSM,("CacheThresholdReached:
    | Volume=%08x\n",Volume));
76080|
76081|     if ( InterlockedIncrement((long
    | *)&DeleteCalledCount)>1 ) {
76082|
    | InterlockedDecrement((long*)&DeleteCalledCount);
76083|         return;
76084|     }
76085|
76086|     __try {
76087|         if ( AcquireOpenCloseResource()==STATUS_WAIT_0
    | ) {
76088|             __try {
76089|                 LARGE_INTEGER Now={0};
76090|                 ULONG At=0,High=0,Used=0;
76091|                 NTSTATUS WarningCode = 0;
76092|                 ULONG
    | Warn,Full,Interval,FullPercent,FullAction;
76093|                 BOOLEAN DeleteAlwaysKeep = FALSE;

```

```

76094|
76095|     | PersistentDictionary::GetCacheThresholds(Volume,Warn,Ful
76096|     | I,Interval,FullPercent,FullAction);
76097|
76098|     | PersistentDictionary::UpdateCacheFileSizes(Volume);
76099|
76100|     | PersistentDictionary::GetVolumeSpaceUsed( Volume,At,
76101|     | High, Used );
76102|     ULONG Percent = (ULONG)(((unsigned
76103|     | __int64)Used * 100) / High);
76104|
76105|     if ( Percent >= Full ) {
76106|         WarningCode =
76107|         | PSM_SECOND_CACHE_THRESHOLD_REACHED;
76108|         if((Percent >= FullPercent) &&
76109|         | (FullAction==CACHE_ACTION_DELETE_ALWAYS_KEEPS)) {
76110|             DeleteAlwaysKeep = TRUE;
76111|         }
76112|     } else {
76113|         if ( Percent >= Warn ) {
76114|             WarningCode =
76115|             | PSM_CACHE_THRESHOLD_REACHED;
76116|         } else {
76117|             // cache file sizes have been
76118|             | updated since the last time we looked..
76119|             try_return(NOTHING);
76120|         }
76121|     }
76122|
76123|     KeQuerySystemTime(&Now);
76124|
76125|     | Debug(DEBUG_DCPSM,("CacheThresholdReached:
76126|     | WarningCode=%08x,
76127|     | LastWarning=%08x\n",WarningCode,TypeOfLastLogEntry));
76128|
76129|     | Debug(DEBUG_DCPSM,("CacheThresholdReached:
76130|     | Now=%016l64x, LastTime=%016l64x, Elapsed=%016l64x,
76131|     | Target=%016l64x\n",
76132|     | Now.QuadPart,
76133|     | TimeOfLastLogEntry.QuadPart,
76134|     | Now.QuadPart -
76135|     | TimeOfLastLogEntry.QuadPart,
76136|     | (unsigned
76137|     | __int64)(SECONDS(Interval*60)) ));
76138|
76139|     if ( ((TypeOfLastLogEntry ==
76140|     | PSM_CACHE_THRESHOLD_REACHED) && (WarningCode ==
76141|     | PSM_SECOND_CACHE_THRESHOLD_REACHED)) ||

```

```

76124|
| Now.QuadPart-TimeOfLastLogEntry.QuadPart >
| SECONDS(Interval*60) ) {
76125|         WCHAR *Strings[3];
76126|         WCHAR Str1[10];
76127|         WCHAR Str2[10];
76128|         WCHAR Str3[10];
76129|         swprintf(Str1,L"%d",Percent);
76130|         swprintf(Str2,L"%d",Warn);
76131|         swprintf(Str3,L"%d",Full);
76132|         Strings[0] = Str1;
76133|         Strings[1] = Str2;
76134|         Strings[2] = Str3;
76135|
76136|         | Debug(DEBUG_DCPSM,("CacheThresholdReached: Logging
| %08x\n",WarningCode));
76137|
76138|         /*lint -save -e740 */
76139|
| LogError((PDEVICE_OBJECT)PSManDriverObject,NULL,WarningC
| ode,0,NULL,0,Strings,3);
76140|         /*lint -restore */
76141|
76142|         TimeOfLastLogEntry = Now;
76143|         TypeOfLastLogEntry = WarningCode;
76144|     }
76145|     if (
| WarningCode==PSM_SECOND_CACHE_THRESHOLD_REACHED ) {
76146|
| FindAndDeleteSnapShotsForVolume(Volume,DeleteAlwaysKeep)
| ;
76147|     }
76148|     try_exit: NOTHING;
76149| } __finally {
76150|     ReleaseOpenCloseResource();
76151| }
76152| } // if OpenCloseResource called
76153| } __finally {
76154|
| InterlockedDecrement((long*)&DeleteCalledCount);
76155| }
76156|
76157| return;
76158| }
76159|
76160| #ifdef DEBUG
76161| void DumpWriteRequest ( tWriteRequest *wr )
76162| {
76163|     Debug(DEBUG_DCPSM,("DumpWriteRequest: wr=%08x

```

```

| -----\n",wr));
76164|  Debug(DEBUG_DCPSM,("DumpWriteRequest:
| DeviceObject = %08x\n", wr->DeviceObject));
76165|  Debug(DEBUG_DCPSM,("DumpWriteRequest:    Irp
| = %08x\n", wr->Irp));
76166|  Debug(DEBUG_DCPSM,("DumpWriteRequest:
| RoundedSector = %016l64x\n",
| wr->RoundedSector.QuadPart));
76167|  Debug(DEBUG_DCPSM,("DumpWriteRequest:
| RoundedCount = %08x\n", wr->RoundedCount));
76168|  Debug(DEBUG_DCPSM,("DumpWriteRequest:
| RealSector = %016l64x\n",
| wr->RealSector.QuadPart));
76169|  Debug(DEBUG_DCPSM,("DumpWriteRequest:
| RealCount = %08x\n", wr->RealCount));
76170| }
76171| #endif /*DEBUG*/
76172|
76173|
76174| // running at IRQL = PASSIVE_LEVEL
76175| // This routine is called when a read for a write has
| completed.
76176| //
76177| /*-----
| -----*/
76178| void SaveOriginalDataThread( PVOID Context )
76179| {
76180|  NTSTATUS Status={0};
76181|  ULONG ThreadNum = (ULONG)Context;
76182|  PFILTERED_EXTENSION DevExt=NULL;
76183|  PIO_STACK_LOCATION NewIrpStackLoc=NULL;
76184|  PIRP Irp=NULL;
76185|  PDEVICE_OBJECT DeviceObject=NULL;
76186|  ULONG ExitThread=0;
76187|  char *Buffer=NULL;
76188|  tWriteRequest *WriteRequest;
76189|  ULONG ThresholdReached=FALSE;
76190|  ULARGE_INTEGER dummy = {0};
76191| #ifdef DEBUG
76192|  // char TempBuffer[50];
76193| #endif
76194|  pkSnapshotEntry p;
76195|
76196|  PAGED_CODE();
76197|
76198|  __try {
76199|  // Debug(DEBUG_THREAD,("Thread %d:
| initing...\n",ThreadNum));
76200|
76201|  ULARGE_INTEGER dummy = {0};

```

```

76202|     SbUpdateThreadStatus( ThreadNum, NULL, NULL,
| _STATE_INIT, dummy, 0, dummy );
76203|
76204|     Status = SbThreadInit( ThreadNum);
76205| }
| __except(ExceptionFilter(GetExceptionInformation())) {
76206|     Status = GetExceptionCode();
76207|     Debug(DEBUG_THREAD,("Thread %d: Exception %08x
| during init\n",ThreadNum,Status));
76208| }
76209|
76210| // Debug(DEBUG_THREAD,("Thread %d: Init done
| %08x\n",ThreadNum,Status));
76211| if ( !INT_SUCCESS(Status) ) {
76212|     goto ExitThreadTerminate;
76213| }
76214|
76215| // Debug(DEBUG_THREAD,("Thread %d: Started T=%08x,
| P=%08x\n",ThreadNum,KeGetCurrentThread(),IoGetCurrentPro
| cess()));
76216|
76217| __try {
76218|     while ( !ExitThread ) {
76219|         ThreadLoop:
76220|         //Debug(DEBUG_THREAD,("Thread %d:
| ThreadLoop\n",ThreadNum));
76221|         TRACE( TRACE_THREADLOOP, 0, 0, 0, 0, 0, "");
76222|
76223|         //Debug(DEBUG_THREAD,("Thread %d: Going to
| sleep Count=%d\n",ThreadNum,ThreadsAwake));
76224|
76225|         ULARGE_INTEGER dummy = {0};
76226|         SbUpdateThreadStatus( ThreadNum,
| DeviceObject, Irp, _STATE_WAITING_FOR_WORK, dummy, 0,
| dummy );
76227|
76228|         Status = SbWaitForThreadWork();
76229|
76230|         //Debug(DEBUG_THREAD,("Thread %d:
| Awake!\n",ThreadNum));
76231|         if ( Status == STATUS_WAIT_1 ) {
76232|             TRACE( TRACE_GETWORK, 0, 0, 0, 0, 0, "");
76233|
76234|             SbUpdateThreadStatus( ThreadNum,
| DeviceObject, Irp, _STATE_WORKING, dummy, 0, dummy );
76235|             //Debug(DEBUG_THREAD,("Thread %d:
| Signaled, Count=%d,
| Awake=%d,Event=%d\n",ThreadNum,pmExamineSemaphore(&Threa
| dSemaphore),ThreadsAwake,KeReadStateEvent(&WorkerThreadE
| vent))););

```

```

76236|
76237|         Irp = SbGetWork( &DeviceObject,
76238|                         &NewIrpStackLoc,
76239|                         &DevExt,
76240|                         &WriteRequest,
76241|                         &Buffer);
76242|
76243|         if ( !Irp ) {
76244|             Debug(DEBUG_THREAD,("Thread %d:
76245| | List is empty\n",ThreadNum));
76246|             goto ThreadLoop;
76247|         }
76248|         // if the read request failed for some
76249|         | reason...
76250|         if ( !NT_SUCCESS(Irp->IoStatus.Status)
76251|         | ) {
76252|             // lets cleanup since we have
76253|             | nothing to do.
76254|             Debug(DEBUG_THREAD,("Thread %d:
76255| | Read failed (%08x)
76256| | Irp=%p\n",ThreadNum,Irp->IoStatus.Status,Irp));
76257|             goto ThreadCleanup;
76258|         }
76259|
76260|         ASSERT(NewIrpStackLoc);
76261|
76262|         SbUpdateThreadStatus(
76263|             ThreadNum,
76264|             DeviceObject,
76265|             Irp,
76266|             _STATE_WORKING,
76267|             | WriteRequest->RealSector,
76268|             | WriteRequest->RealCount,
76269|             | WriteRequest->RealSector );
76270|
76271|         TRACE(
76272|             TRACE_PROCESSECTOR,
76273|             0,
76274|             WriteRequest->RealSector.HighPart,
76275|             WriteRequest->RealSector.LowPart,
76276|             WriteRequest->RealCount,
76277|             "");
76278|
76279|         /*
76280|         We acquire the snapshot resource

```

```

    | shared denying
76277|         writers (basically just
    | add/removing of snapshots)
76278|         The reason we do a starve, is that
    | io may occur
76279|         while some thread has the resource
    | acquired shared,
76280|         and then a writer is waiting for
    | access. Normally, the
76281|         shared reader is put to sleep until
    | all shared
76282|         access is blocked, and the writer
    | can run and
76283|         release it. In cases when we need
    | to do io (ie
76284|         mounting the volume) while the
    | resource is acquired
76285|         shared, unless we want to block, we
    | need to go ahead
76286|         and do the io
76287|
76288|         */
76289|         KeEnterCriticalRegion();
76290|
    | ExAcquireSharedStarveExclusive(&GlobalData->SnapShotReso
    | urce, TRUE);
76291|         KeLeaveCriticalRegion();
76292|
76293| //         GetSnapShotForRead();
76294|         __try {
76295|             tDictSiblingInfo Info;
76296|             memset(&Info,0,sizeof(Info));
76297|
76298|             p =
    | GetTopSnapShot(&DevExt->SnapShots);
76299|             while ( p ) {
76300|                 ULARGE_INTEGER UL;
76301|                 ULARGE_INTEGER DS;
76302|                 ULONG Did;
76303|
76304|                 UL.QuadPart =
    | WriteRequest->RoundedSector.QuadPart;
76305|                 DS.QuadPart =
    | WriteRequest->RoundedCountInBytes.QuadPart;
76306|
76307|                 //Debug(DEBUG_THREAD,("Thread
    | %d: Writing\n",ThreadNum));
76308|                 SbUpdateThreadStatus(
    | ThreadNum, DeviceObject, Irp, _STATE_WAITING_FOR_WRITE,
    | WriteRequest->RealSector, WriteRequest->RealCount,

```

```

    | WriteRequest->RealSector );
76309|
76310|         Status =
    | p->Dictionary->searchAndInsertMultiple(DevExt,UL,WriteRe
    | quest->RoundedCount,Did,NULL,DS,Buffer,&Info,0);
76311|
76312|         if (
    | ((pPersistentDictionary)(p->Dictionary))->IsCacheWarning
    | ThresholdReached() ) {
76313|             ThresholdReached=TRUE;
76314|         }
76315|         if ( Status ==
    | STATUS_END_OF_FILE ) {
76316|             // told to stop scanning
    | the snapshots...
76317|             Status = STATUS_SUCCESS;
76318|             DoneWithSnapShot(p);
76319|             break;
76320|         }
76321|
76322|         if ( !NT_SUCCESS(Status) ) {
76323|             FailRequest(p,Status);
76324|         }
76325|
76326|         p =
    | GetNextSnapShot(&DevExt->SnapShots,p);
76327|     } // while(p)
76328| } __finally {
76329|     ReleaseSnapShotForRead();
76330| }
76331|
76332| ThreadCleanup:
76333| TRACE(
76334|     TRACE_THREADCLEANUP,
76335|     0,
76336|     WriteRequest->RealSector.HighPart,
76337|     WriteRequest->RealSector.LowPart,
76338|     WriteRequest->RealCount,
76339|     "");
76340|
76341|     SbUpdateThreadStatus( ThreadNum,
    | DeviceObject, Irp, _STATE_CLEANUP,
    | WriteRequest->RealSector, WriteRequest->RealCount,
    | dummy );
76342|     // cleanup this irp.
76343|
76344| #if 0
76345| #ifdef DEBUG
76346|     // This was added 2001 Nov 07 to help
    | find revert ntfs corruption.

```



```

76347|         if (
| WriteRequest->RoundedSector.QuadPart < 0x800 ) {
76348|         | Debug(DEBUG_THREAD,("SaveOriginalDataThread: Old data
| granule follows (granule=%016l64x)\n",
76349|         | WriteRequest->RoundedSector.QuadPart /
| (GRANULE_SIZE/512) ));
76350|
76351|         DumpSector ( (char
| *) (WriteRequest->Buffer), GRANULE_SIZE );
76352|
76353|         char *KernelPointer = (char
| *) MmGetSystemAddressForMdlSafe(
| WriteRequest->Irp->MdlAddress, NormalPagePriority );
76354|         | Debug(DEBUG_THREAD,("SaveOriginalDataThread: granule we
| are about WRITE (granule=%016l64x)\n",
76355|         | WriteRequest->RoundedSector.QuadPart /
| (GRANULE_SIZE/512) ));
76356|
76357|         DumpWriteRequest (WriteRequest);
76358|
76359|         DumpSector ( KernelPointer,
| WriteRequest->ByteLength );
76360|     }
76361|     #endif /*DEBUG*/
76362| #endif
76363|
76364|         //Debug(DEBUG_THREAD,("Thread %d:
| Freeing pool %p\n",ThreadNum,WriteRequest->Buffer));
76365|         /*lint -save -e613 */
76366|         | MemFreePool((PVOID)(WriteRequest->Buffer));
76367|         /*lint -restore */
76368|         WriteRequest->Buffer = NULL;
76369|
76370|         //Debug(DEBUG_THREAD,("Thread %d:
| Freeing Irp %p\n",ThreadNum,Irp));
76371|         // save the status since we freeing the
| request
76372|         Status = Irp->IoStatus.Status;
76373|
76374|         IrpFreeIrp(Irp);
76375|
76376|         KIRQL oldIrql;
76377|         | pmAcquireSpinLock(&WriteSpinLock,&oldIrql);
76378|

```

```

    | RemoveEntryList(&(WriteRequest->ProcessingEntry));
76379|
    | pmReleaseSpinLock(&WriteSpinLock,oldIrql);
76380|
76381|         if(NT_SUCCESS(Status)) {
76382|             InterlockedDecrement(
    | (PLONG)&OutstandingRequests );
76383| #ifdef DEBUG
76384|             // so we know when the io completed
76385|             (void)PSManSendOrigWrite(
    | WriteRequest->DeviceObject, WriteRequest->Irp);
76386| #else
76387|             (void)PSManPassThru(
    | WriteRequest->DeviceObject, WriteRequest->Irp);
76388| #endif
76389|         } else {
76390|             // nothing to do as the io has
    | already been completed in the completion routine for
    | the
76391|             // read
    | (OtManWriteCompletionDevice).
76392|         }
76393|
76394|         FREE_POINTER(WriteRequest);
76395|
76396|         //Debug(DEBUG_THREAD,("Thread %d: Done
    | processing Sector %d, %d
    | (Irp=%p)\n",ThreadNum,Sector,Count,Irp));
76397|         // change the threshold after we have
    | released the write we
76398|         // were processing so we dont deadlock.
76399|         if ( ThresholdReached ) {
76400|
    | CacheThresholdReached(DeviceObject);
76401|         }
76402|
76403|     } else {
76404|         // most likely STATUS_WAIT_0 but could
    | be an error condition
76405|         //Debug(DEBUG_THREAD,("Thread %d:
    | Error! %08x on wait\n",ThreadNum,Status));
76406|         FailRequest(NULL,PSM_CANCELED_BY_USER);
76407|         ExitThread = 1;
76408|     }
76409| } // while(!ExitThread)
76410| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
76411|     Status=GetExceptionCode();
76412|     Debug(DEBUG_THREAD,("Thread %d: Error!
    | Exception %08x\n",ThreadNum, Status));

```

```

76413|     FailRequest(NULL,Status);
76414| }
76415| //ExitThreadJump:
76416|
76417| TRACE( TRACE_THREADEXIT, 0, 0, 0, 0, "");
76418|
76419| //Debug(DEBUG_THREAD,("Thread %d: CurrentIrql =
    | %d\n",ThreadNum, KeGetCurrentIrql()));
76420| SbUpdateThreadStatus( ThreadNum, NULL, NULL,
    | _STATE_DEINIT, dummy, 0, dummy );
76421|
76422|
76423| ExitThreadTerminate:
76424|
76425| SbDecrementRunningThreads();
76426|
76427| Debug(DEBUG_THREAD,("Thread %d:
    | Exiting\n",ThreadNum));
76428| PsTerminateSystemThread( 0 );
76429| }
76430|
76431| /*-----
    | -----*/
76432| STATIC int SbMakeAnotherThread ( void )
76433| {
76434|     HANDLE TempHandle=NULL;
76435|     NTSTATUS ntStatus=0;
76436|     ULONG ThreadNum=0;
76437|     ULONG Save=0;
76438|
76439|     TRACE( TRACE_MAKETHREAD, 0, 0, 0, 0, "");
76440|
76441|     pmAcquireMutex ( &WorkerThreadMutex, NULL );
76442|
76443|     // cant do much at APC_LEVEL which fast mutexes
    | move us to...
76444|
76445|     Save = GlobalThreadCount;
76446|
76447|     ThreadNum=NumberOfThreads;
76448|
76449|     if ( NumberOfThreads<MaxThreads ) {
76450|         // inc first or thread will stop with invalid
    | thread count
76451|         NumberOfThreads++;
76452|         ntStatus=STATUS_SUCCESS;
76453|     } else {
76454|         // if at max, then dont create any more
76455|         ntStatus = STATUS_UNSUCCESSFUL;
76456|     }

```

```

76457|
76458| pmReleaseMutex ( &WorkerThreadMutex );
76459|
76460| if ( NT_SUCCESS(ntStatus) ) {
76461|     ntStatus = pmStartThread(
76462|         | (PKSTART_ROUTINE)SaveOriginalDataThread, // IN
76463|         | PKSTART_ROUTINE StartRoutine,
76464|         | (PVOID)ThreadNum,
76465|         | // IN PVOID StartContext
76466|         | &TempHandle
76467|         | // OUT PHANDLE ThreadHandle,
76468|         );
76469|     if ( NT_SUCCESS(ntStatus) ) {
76470|         ntStatus = ObReferenceObjectByHandle(
76471|             | TempHandle, // IN HANDLE Handle,
76472|             | THREAD_ALL_ACCESS, // IN ACCESS_MASK DesiredAccess,
76473|             | NULL,
76474|             | // IN POBJECT_TYPE ObjectType, /*
76475|             | optional */
76476|             | (KPROCESSOR_MODE)KernelMode, // IN
76477|             | KPROCESSOR_MODE AccessMode,
76478|             | &ThreadObjects[ThreadNum].ThreadObject, // OUT PVOID
76479|             | *Object,
76480|             | NULL
76481|             | // OUT POBJECT_HANDLE_INFORMATION HandleInformation
76482|             | /* optional */
76483|             );
76484|         Debug(DEBUG_THREAD,("Thread %d Handle =
76485|             | %08x, Object=%08x,
76486|             | Status=%08x\n",ThreadNum,TempHandle,ThreadObjects[Thread
76487|             | Num].ThreadObject,ntStatus));
76488|         // Dont need the handle anymore now that we
76489|         | have an object
76490|         ZwClose(TempHandle);
76491|         TempHandle = NULL;
76492|     }
76493|
76494|     // if thread was started, wait for it to start.
76495|     if ( NT_SUCCESS(ntStatus) ) {
76496|         LARGE_INTEGER TimeToWait;
76497|
76498|         ASSERT(KeGetCurrentIrql() <
76499|             | DISPATCH_LEVEL);

```

```

76488|
76489|         //Debug(DEBUG_THREAD,("Writer: GTC=%d,
| Save=%d\n",GlobalThreadCount,Save));
76490|         TimeToWait.QuadPart =
| RELATIVE(MICROSECONDS(10));
76491|         ULARGE_INTEGER dummy = {0};
76492|         SbUpdateWriterStatus(
| _STATE_WAITING_FOR_EVENT, dummy, 0 );
76493|         while ( GlobalThreadCount == Save ) {
76494|             //Debug(DEBUG_THREAD,("Writer: GTC=%d,
| Save=%d\n",GlobalThreadCount,Save));
76495|
76496|             KeDelayExecutionThread(
76497|
| (KPROCESSOR_MODE)KernelMode, // IN KPROCESSOR_MODE
| WaitMode,
76498|
| FALSE, // IN
| BOOLEAN Alertable,
76499|
| &TimeToWait //
| IN PLARGE_INTEGER Interval
76500|
| );
76501|     }
76502|     /*
76503|         ASSERT(KeGetCurrentIrql() <
| DISPATCH_LEVEL);
76504|         pmWaitForSingleObject(
| &WorkerThreadEvent,NULL);
76505|     */
76506|     } else {
76507|         pmAcquireMutex ( &WorkerThreadMutex, NULL
| );
76508|         NumberOfThreads--;
76509|         pmReleaseMutex ( &WorkerThreadMutex );
76510|     }
76511| }
76512|
76513| return NT_SUCCESS(ntStatus) ? TRUE : FALSE;
76514| }
76515|
76516|
76517| /*-----
| -----*/
76518| STATIC NTSTATUS SbWaitForWriteFromNT()
76519| {
76520|     PVOID ObjectTable[2] = {
| &PSManExitingEvent, &WriteSemaphore};
76521|
76522|     TRACE( TRACE_WAITWRITEFROMNT, 0, 0, 0, 0, "");
76523|     ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
76524|     return

```

```

    | pmWaitForMultipleObjects(ObjectTable,2,NULL);
76525| }
76526|
76527|
76528| /*-----
    | -----*/
76529| NTSTATUS SbCompleteWritesOnQueue()
76530| {
76531|     PLIST_ENTRY    ListEntry=NULL;
76532|     PDEVICE_OBJECT DeviceObject=NULL;
76533|     PIRP           Irp=NULL;
76534|     tWriteRequest  *WriteRequest=NULL;
76535|
76536|     TRACE( TRACE_COMPLETEWRITESONQUEUE, 0, 0, 0, 0,
    | "");
76537|     while ( !IsListEmpty(&WriteQueue) ) {
76538|         ListEntry = ExInterlockedRemoveHeadList (
76539|
    | &WriteQueue,    // List Head
76540|
    | &WriteSpinLock // Lock
76541|
    | );
76542|
76543|         if ( (ListEntry) && (ListEntry!=&WriteQueue) )
    | {
76544|             /*lint -save -e413 */
76545|             WriteRequest = CONTAINING_RECORD(
    | ListEntry, tWriteRequest, ListEntry );
76546|             /*lint -restore */
76547|             DeviceObject = WriteRequest->DeviceObject;
76548|             Irp          = WriteRequest->Irp;
76549|
76550|             // If its on the write queue, its not on
    | the processing queue
76551|
76552|             FREE_POINTER(WriteRequest);
76553| #if 1
76554|             File_PrintIrp("Writer Cleanup:
    | ",DeviceObject,Irp);
76555| #endif
76556|             InterlockedDecrement(
    | (PLONG)&OutstandingRequests );
76557|             (void)PSManPassThru(DeviceObject,Irp);
76558|         } else {
76559|             Debug(DEBUG_THREAD,("Writer Cleanup: Error
    | ListEntry is empty\n"));
76560|         }
76561|     }
76562|     return STATUS_SUCCESS;
76563| }

```

```

76564|
76565| NTSTATUS SbCompleteWritesReadQueue()
76566| {
76567|     PLIST_ENTRY    ListEntry=NULL;
76568|     PDEVICE_OBJECT DeviceObject=NULL;
76569|     PIRP           Irp=NULL;
76570|     tWriteRequest  *WriteRequest=NULL;
76571|
76572|     while ( !IsListEmpty(&WriteAfterReadQueue) ) {
76573|         ListEntry = ExInterlockedRemoveHeadList (
76574|             | &WriteAfterReadQueue,    // List Head
76575|             | &WriteAfterReadSpinLock  // Lock
76576|             );
76577|
76578|         if ( (ListEntry) &&
76579|             | (ListEntry!=&WriteAfterReadQueue) ) {
76580|             /*lint -save -e413 */
76581|             WriteRequest = CONTAINING_RECORD(
76582|                 | ListEntry, tWriteRequest, ListEntry );
76583|             /*lint -restore */
76584|             DeviceObject = WriteRequest->DeviceObject;
76585|             Irp          = WriteRequest->Irp;
76586|             FREE_POINTER(WriteRequest);
76587|             #if 1
76588|             File_PrintIrp("WriteAfterRead Cleanup:
76589|                 | ",DeviceObject,Irp);
76590|             #endif
76591|             InterlockedDecrement(
76592|                 | (PLONG)&OutstandingRequests );
76593|             (void)PSManPassThru(DeviceObject,Irp);
76594|             } else {
76595|                 Debug(DEBUG_THREAD,("WriteAfterRead
76596|                     | Cleanup: Error ListEntry is empty\n"));
76597|             }
76598|         }
76599|     }
76600|     return STATUS_SUCCESS;
76601| }
76602|
76603| /*-----*/
76604| | -----*/
76605| STATIC NTSTATUS SbWaitForFreeThread( )
76606| {
76607|     NTSTATUS Status;
76608|     LARGE_INTEGER CurrentTime={0};
76609|     LARGE_INTEGER EndTime={0};
76610|
76611|     WaitForThreads:

```

```

76606|   Status = SbThreadQueueWait();
76607|
76608|   if ( Status == STATUS_TIMEOUT ) {
76609|       //Debug(DEBUG_THREAD,("Writer: Creating another
       | Thread\n"));
76610|       if ( SbMakeAnotherThread() ) {
76611|           Debug(DEBUG_THREAD,("Writer: Created
       | another Thread\n"));
76612|       } else {
76613|           //Debug(DEBUG_THREAD,("Writer: Error!
       | unable to make another thread\n"));
76614|
76615|           // To keep hangs from occurring, we will see
       | if we are hung for
76616|           // longer than x number of seconds, if so,
       | abort the backup.
76617|
76618|           if ( EndTime.QuadPart == 0 ) {
76619|               KeQuerySystemTime( &EndTime );
76620|               EndTime.QuadPart +=
       | MICROSECONDS(gHungSystemTimeOut);
76621|           }
76622|           KeQuerySystemTime( &CurrentTime );
76623|           if ( CurrentTime.QuadPart >
       | EndTime.QuadPart ) {
76624|               Debug(DEBUG_THREAD,("Writer: Error!
       | Timeout while waiting for threads\n"));
76625| //           DbgBreakPoint();
76626|               FailRequest(NULL,PSM_ERROR_DEADLOCK);
76627|               SbCompleteWritesOnQueue();
76628|               return PSM_ERROR_DEADLOCK;
76629|           }
76630|       }
76631|       goto WaitForThreads;
76632|   }
76633|
76634|   return Status;
76635| }
76636|
76637| /*-----
       | -----*/
76638| STATIC NTSTATUS SbAllocateIrpResources ( PCHAR *Buffer,
76639|                                         PIRP *Irp,
76640|                                         PMDL *Mdl,
76641|                                         USHORT
       | StackSize,
76642|                                         ULONG
       | ByteCount
76643|                                         )
76644| {

```



```

76645| NTSTATUS Status = STATUS_INSUFFICIENT_RESOURCES;
76646|
76647| TRACE( TRACE_ALLOCATERESOURCES, 0, 0, ByteCount, 0,
| "");
76648| // allocate memory on aligned boundery
76649| (*Buffer) = (PCHAR)
| MemAllocatePoolWithTag(PagedPoolCacheAligned,
| ByteCount, BUFFTAG);
76650|
76651| if ( (*Buffer) ) {
76652|     (*Irp) = IrpAllocateIrp ( StackSize );
76653|     if ( (*Irp) ) {
76654|         (*Mdl) = IoAllocateMdl(
| (*Buffer),ByteCount,FALSE,FALSE,(*Irp));
76655|         if ( (*Mdl) ) {
76656|             /*lint -save -e149 */
76657|             __try {
76658|                 //MmBuildMdlForNonPagedPool(Mdl);
76659|                 MmProbeAndLockPages((*Mdl),
| (KPROCESSOR_MODE)KernelMode,IoModifyAccess);
76660|                 return STATUS_SUCCESS;
76661|             } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
76662|                 Debug(DEBUG_THREAD,"Exception
| while locking Mdl %p for %p\n",(*Mdl),(*Buffer));
76663|                 Status = GetExceptionCode();
76664|             }
76665|             /*lint -restore */
76666|             IoFreeMdl((*Mdl));
76667|         }
76668|         IrpFreeIrp((*Irp));
76669|     }
76670|     MemFreePool((*Buffer));
76671|     *Buffer=NULL;
76672| }
76673|
76674| return Status;
76675| }
76676|
76677| /*-----
| -----*/
76678| STATIC NTSTATUS SbAllocateIrpResourcesOrWait ( PCHAR
| *Buffer,
76679|
| PIRP
| *Irp,
76680|
| PMDL
| *Mdl,
76681|
| USHORT
| StackSize,
76682|
| ULONG

```

```

    | ByteCount
76683|         )
76684| {
76685|     NTSTATUS Status = STATUS_INSUFFICIENT_RESOURCES;
76686|
76687|     while ( Status==STATUS_INSUFFICIENT_RESOURCES ) {
76688|         Status = SbAllocateIrpResources( Buffer,
76689|                                         Irp,
76690|                                         Mdl,
76691|                                         StackSize,
76692|                                         ByteCount
76693|                                         );
76694|         if ( Status == STATUS_INSUFFICIENT_RESOURCES )
76695|         | {
76696|             LARGE_INTEGER TimeOut;
76697|             TimeOut.QuadPart =
76698|             | RELATIVE(MICROSECONDS(((signed
76699|             | long)NewThreadStartDelay)));
76700|             // if not exiting, wait for some time, and
76701|             | try alloc again
76702|             Debug(DEBUG_THREAD,("Waiting for resources
76703|             | to become available\n"));
76704|             ASSERT(KeGetCurrentIrql() <
76705|             | DISPATCH_LEVEL);
76706|             Status = pmWaitForSingleObject(
76707|             | &PSManExitingEvent,&TimeOut );
76708|             // if the timeout occurred, try allocating
76709|             | resources again
76710|             // else, its time to exit...
76711|             if ( Status==STATUS_TIMEOUT ) {
76712|                 Status = STATUS_INSUFFICIENT_RESOURCES;
76713|             }
76714|         }
76715|     }
76716|     return Status;
76717| }
76718|
76719| /*-----*/
76720| | -----*/
76721| STATIC NTSTATUS SbInitReadIrp(
76722|     PIRP      NewIrp,
76723|     PMDL      Mdl,
76724|     PETHREAD   Thread,
76725|     PCHAR
76726|     | AuxiliaryBuffer,
76727|     PFILE_OBJECT

```

```

    | OriginalFileObject
76722|         )
76723| {
76724|     // set up our IRP to be sent down to the lower
    | driver.
76725|
76726|     NewIrp->MdlAddress          = Mdl;
76727|     NewIrp->AssociatedIrp.SystemBuffer = NULL;
76728|
76729|     // for removable media requests so it can display a
    | dialog box.
76730|     NewIrp->Tail.Overlay.Thread      = Thread;
76731|     NewIrp->RequestorMode            =
    | (KPROCESSOR_MODE)KernelMode;
76732|
76733|     // misc parameters that i dont know if there needed
    | but look like
76734|     // good candidates on saving
76735|     // undocumented fields...
76736|     NewIrp->Tail.Overlay.AuxiliaryBuffer =
    | AuxiliaryBuffer;
76737|     NewIrp->Tail.Overlay.OriginalFileObject =
    | OriginalFileObject;
76738|
76739|     // you also have the following fields i dont know
    | if they are
76740|     // needed or not..
76741|     // NewIrp->UserIoSb
76742|     // NewIrp->UserEvent
76743|
76744|     NewIrp->UserBuffer = NULL;
76745|
76746|     // IoBuildSynchronousFsdRequest you pass in
76747|     // IN PKEVENT Event, and OUT PIO_STATUS_BLOCK IoSb
76748|     // so this makes me think i should set the fields
    | above...
76749|
76750|     // Indicate that this is a READ operation.
76751|     NewIrp->Flags          =
    | IRP_READ_OPERATION;
76752|
76753|     NewIrp->IoStatus.Status      =
    | STATUS_PENDING;
76754|     NewIrp->IoStatus.Information = 0;
76755|
76756|     return STATUS_SUCCESS;
76757| }
76758|
76759|
76760| /*-----

```

```

| -----*/
76761| STATIC NTSTATUS SblInitOtherIrpStack (
76762|             PIO_STACK_LOCATION
| Stack,
76763|             PDEVICE_OBJECT
| DeviceObject,
76764|             PFILE_OBJECT
| FileObject,
76765|             UCHAR
| Flags,
76766|             PVOID
| Arg1,
76767|             PVOID
| Arg2,
76768|             PVOID
| Arg3,
76769|             PVOID
| Arg4
76770|             )
76771| {
76772|     Stack->CompletionRoutine = NULL;
| // set by SetIoCompletionRoutine
76773|     Stack->Context          = 0;
| // set by SetIoCompletionRoutine
76774|
76775|     Stack->Flags            = Flags;
| // Read/Write - SL_OVERRIDE_VERIFY_VOLUME, etc...
76776|     Stack->Control          = 0;
| // SL_PENDING_RETURNED
76777|     Stack->MajorFunction     = IRP_MJ_READ;
76778|     Stack->MinorFunction     = 0;
76779|     Stack->DeviceObject      = DeviceObject;
76780|     Stack->FileObject        = FileObject;
76781|
76782|     Stack->Parameters.Others.Argument1 = Arg1;
76783|     Stack->Parameters.Others.Argument2 = Arg2;
76784|     Stack->Parameters.Others.Argument3 = Arg3;
76785|     Stack->Parameters.Others.Argument4 = Arg4;
76786|     return STATUS_SUCCESS;
76787| }
76788|
76789| /*-----*/
| -----*/
76790| STATIC NTSTATUS SblInitReadIrpStack(
76791|             PIO_STACK_LOCATION
| Stack,
76792|             PDEVICE_OBJECT
| DeviceObject,
76793|             PFILE_OBJECT
| FileObject,

```

```

76794|                UCHAR
    | Flags,
76795|                PLARGE_INTEGER
    | ByteOffset,
76796|                ULONG
    | Length,
76797|                ULONG
    | Key
76798|                )
76799| {
76800|     Stack->CompletionRoutine    = NULL;
    | // set by SetIoCompletionRoutine
76801|     Stack->Context              = 0;
    | // set by SetIoCompletionRoutine
76802|
76803|     Stack->Flags                = Flags;
    | // Read/Write - SL_OVERRIDE_VERIFY_VOLUME, etc...
76804|     Stack->Control              = 0;
    | // SL_PENDING_RETURNED
76805|     Stack->MajorFunction        = IRP_MJ_READ;
76806|     Stack->MinorFunction        = 0;
76807|     Stack->DeviceObject         = DeviceObject;
76808|     Stack->FileObject           = FileObject;
76809|
76810|     Stack->Parameters.Read.Length    = Length;
76811|     Stack->Parameters.Read.ByteOffset = *ByteOffset;
76812|     Stack->Parameters.Read.Key       = Key;
    | // this may cause trouble later on.....
76813|     return STATUS_SUCCESS;
76814| }
76815|
76816|
76817| // running at IRQL = PASSIVE_LEVEL
76818| //
76819| // repeat
76820| //     Wait For NT Write
76821| //     Wait for Free Thread
76822| //     Process Read Old Data
76823| // until Time To Exit
76824| //
76825| // Read Old Data Completed
76826| //     Send Original Write to Lower Level Driver
76827| //     Add Read Irp to Thread Queue
76828| //     Signal Thread Queue
76829| //
76830| // Thread Queue
76831| //     Init
76832| //     repeat
76833| //         Wait For Signal
76834| //         If not duplicate

```

```

76835| //      Write Old Data to cache WAIT
76836| //      Add to Tree
76837| //      Cleanup Read Irp
76838| //  until Time To Exit
76839| //  Cleanup
76840| //
76841| //
76842| // 1. Takes a write from the NT Dispatcher.
76843| // 2. Waits for a thread
76844| // 3. Then composes a Read Irp, and
76845| // 4. Sends it to the lower level driver.
76846| // 5. Loops back to 1
76847|
76848| // This thread is awoken when the Write dispatch
    | (write.c) has realized
76849| // that we need to process this request. This has the
    | side affect of
76850| // serializing all writes.
76851| /*-----
    | -----*/
76852| void WriteDispatchThread ( PVOID Context )
76853| {
76854|     NTSTATUS      Status={0};
76855|     PFILTERED_EXTENSION DevExt=NULL;
76856|     ULONG          ExitThread=0;
76857|     PIO_STACK_LOCATION currentIrpStack=NULL;
76858|     PIO_STACK_LOCATION nextIrpStack=NULL;
76859|     PIO_STACK_LOCATION NewIrpCurrentStack=NULL;
76860|     PIRP           NewIrp=NULL;
76861|     PCHAR          buffer=NULL;
76862|     PMDL           Mdl=NULL;
76863|     KIRQL          oldIrql=0;
76864|     PLIST_ENTRY     ListEntry=NULL;
76865|     tWriteRequest   *WriteRequest=NULL;
76866|
76867|     NOT_REFERENCED(Context);
76868|
76869|     PAGED_CODE();
76870|
76871|     Debug(DEBUG_THREAD,("Writer: Started T=%08x,
    | P=%08x,
    | irql=%d\n",KeGetCurrentThread(),IoGetCurrentProcess(),Ke
    | GetCurrentIrql()));
76872|     ULARGE_INTEGER dummy = {0};
76873|     SbUpdateWriterStatus( _STATE_INIT, dummy, 0 );
76874|
76875|     SbIncrementRunningThreads();
76876|
76877| #ifndef SYNC
76878|     // make sure other threads are ready first...

```

```

76879|   if ( !NT_SUCCESS(SbWaitOnEvent( &Thread0Inited )) )
76880|   {
76881|       goto ExitThreadJump;
76882|   }
76883| #endif
76884|
76885| // threads normally start with variable priority.
76886| | This sets it to the
76887| // lowest of the time critical prioritys
76888| // Realtime threads have no quantum timeout. They
76889| | only give up the
76890| // cpu when they voluntarily go into a wait state,
76891| | or when preempted by
76892| // a thread of higher priority
76893| //KeSetPriorityThread( KeGetCurrentThread(),
76894| | LOW_REALTIME_PRIORITY );
76895| Debug(DEBUG_THREAD,("Writer: Waiting for
76896| | work...\n"));
76897|
76898| __try {
76899|     while ( !ExitThread ) {
76900|         ThreadLoop:
76901|         //Debug(DEBUG_THREAD,("Writer: Going to
76902| | sleep irq1=%d\n",KeGetCurrentIrql()));
76903|         SbUpdateWriterStatus(
76904| | _STATE_WAITING_FOR_WORK, dummy, 0 );
76905|
76906|         Status = SbWaitForWriteFromNT();
76907|
76908|         if ( Status == STATUS_WAIT_1 ) {
76909|             //Debug(DEBUG_THREAD,("Writer: write
76910| | from nt, irq1=%d\n",KeGetCurrentIrql()));
76911|
76912|             SbUpdateWriterStatus(
76913| | _STATE_WAITING_FOR_THREAD, dummy, 0 );
76914|
76915|             // returns 0 for success, else error
76916|             | code (STATUS_WAIT_1 if exiting)
76917|             // PSM_ERROR_DEADLOCK if system is
76918|             | hung...
76919|             #if 1
76920|                 Status = SbWaitForFreeThread();
76921|             #else
76922|                 Status = 0;
76923|             #endif
76924|
76925|             if ( Status == 0 ) {
76926|                 //Debug(DEBUG_THREAD,("Writer:
76927| | Thread is free, irq1=%d\n",KeGetCurrentIrql()));
76928|                 TRACE( TRACE_WORKERGET, 0, 0, 0, 0,

```

```

| "");
76916|          //Debug(DEBUG_THREAD,("Writer:
| Signaled %d, sema=%d\n",ThreadsAwake,GetState()));
76917|
76918|          //Debug(DEBUG_THREAD,("Writer: Done
| with trace, irq!=%d\n",KeGetCurrentIrql()));
76919|          SbUpdateWriterStatus(
| _STATE_WORKING, dummy, 0 );
76920|          GetAnother:
76921|          pmAcquireSpinLock ( &WriteSpinLock,
| &oldIrql );
76922|          WriteQueueDepth--;
76923|          ListEntry = RemoveHeadList
| (&WriteQueue);
76924|          pmReleaseSpinLock( &WriteSpinLock,
| oldIrql );
76925|          //Debug(DEBUG_THREAD,("Writer:
| resetting irq! from=%d back to
| %d\n",KeGetCurrentIrql(),oldIrql));
76926|
76927|          if ( (!ListEntry) ||
| (ListEntry==&WriteQueue) ) {
76928|              Debug(DEBUG_THREAD,("Writer:
| Error ListEntry is empty\n"));
76929|              SbThreadQueueDec(); //
| Decrement the increment we did, since we did not start
| a thread
76930|              goto ThreadLoop;
76931|          }
76932|
76933|          //Debug(DEBUG_THREAD,("Writer: Got
| Work, irq!=%d\n",KeGetCurrentIrql()));
76934|
76935|          /*lint -save -e413 */
76936|          WriteRequest = CONTAINING_RECORD(
| ListEntry, tWriteRequest, ListEntry );
76937|          /*lint -restore */
76938|          //Debug(DEBUG_THREAD,("Writer:
| WriteRequest=%08x, DeviceObject=%08x,
| Irp=%08x\n",WriteRequest,DeviceObject,Irp));
76939|
76940|          if (
| IsBeingProcessedEx(WriteRequest) ) {
76941|              BOOLEAN OnlyOne=FALSE;
76942|
76943|              pmAcquireSpinLock (
| &WriteSpinLock, &oldIrql );
76944|              WriteQueueDepth++;
76945|              InsertTailList
| (&WriteQueue,&WriteRequest->ListEntry);

```



```

76946|
76947|           // if only one on list
76948|           if ( WriteQueue.Flink ==
    | &WriteRequest->ListEntry ) {
76949|               OnlyOne=TRUE;
76950|           }
76951|           pmReleaseSpinLock(
    | &WriteSpinLock, oldIrql );
76952|           if ( OnlyOne ) {
76953|               LARGE_INTEGER TimeToWait =
    | {0};
76954|               TimeToWait.QuadPart =
    | RELATIVE(MILLISECONDS(1));
76955|               KeDelayExecutionThread(
    | (KPROCESSOR_MODE)KernelMode, FALSE, &TimeToWait );
76956|           }
76957|           goto GetAnother;
76958|       }
76959|
76960|
76961|       DevExt =
    | GetFilteredExtension(WriteRequest->DeviceObject);
76962|       currentIrpStack =
    | IoGetCurrentIrpStackLocation(WriteRequest->Irp);
76963|       nextIrpStack =
    | IoGetNextIrpStackLocation(WriteRequest->Irp);
76964|
76965|       SbUpdateWriterStatus(
    | _STATE_WAITING_FOR_MEMORY, WriteRequest->RealSector,
    | WriteRequest->RealCount);
76966|
76967|       | ASSERT(WriteRequest->RoundedCountInBytes.LowPart %
    | GRANULE_SIZE == 0);
76968|
76969|       Status =
    | SbAllocateIrpResourcesOrWait( &buffer,
76970|       | &NewIrp,
76971|       | &Mdl,
76972|       | WriteRequest->DeviceObject->StackSize,
76973|       | WriteRequest->RoundedCountInBytes.LowPart
76974|       | );
76975|
76976|       if ( !NT_SUCCESS(Status) ) {
76977|

```

```

    | Debug(DEBUG_THREAD|DEBUG_ERROR,("Error! %08x while
    | allocating resources\n",Status));
76978|         FailRequest(NULL,Status);
76979|         Debug(DEBUG_THREAD |
    | DEBUG_INFO,("Calling original driver with orginal irp
    | %p\n",WriteRequest->Irp));
76980|         SbThreadQueueDec(); //
    | Decrement the increment we did, since we did not start
    | a thread
76981|         InterlockedDecrement(
    | (PLONG)&OutstandingRequests );
76982|
    | pmAcquireSpinLock(&WriteSpinLock,&oldIrql);
76983|
    | RemoveEntryList(&(WriteRequest->ProcessingEntry));
76984|
    | pmReleaseSpinLock(&WriteSpinLock,oldIrql);
76985|         Status = PManPassThru(
    | WriteRequest->DeviceObject, WriteRequest->Irp );
76986|         MemFreePool(WriteRequest);
76987|         goto ThreadLoop;
76988|     }
76989|
76990|         TRACE(
76991|             TRACE_SETTINGUP,
76992|             0,
76993|
    | WriteRequest->RealSector.HighPart,
76994|
    | WriteRequest->RealSector.LowPart,
76995|             WriteRequest->RealCount,
76996|             "");
76997|
76998| #ifdef LINT
76999|         if ( !NewIrp ) goto ThreadLoop;
77000| #endif
77001|
77002|         ASSERT(NewIrp);
77003|         ASSERT(Mdl);
77004|         ASSERT(buffer);
77005|
77006|         WriteRequest->Buffer = buffer;
77007|
77008|         // push stack location so it points
    | at the first valid slot
77009|         // we use this slot to hold our
    | contexts
77010|         IoSetNextIrpStackLocation( NewIrp
    | );
77011|         NewIrpCurrentStack =

```

```

    | IoGetCurrentIrpStackLocation(NewIrp);
77012|
77013|         SbUpdateWriterStatus(
    | _STATE_WORKING, WriteRequest->RealSector,
    | WriteRequest->RealCount);
77014|
77015|         SbInitReadIrp( NewIrp,
77016|                        Mdl,
77017|
    | WriteRequest->Irp->Tail.Overlay.Thread,
77018|
    | WriteRequest->Irp->Tail.Overlay.AuxiliaryBuffer,
77019|
    | WriteRequest->Irp->Tail.Overlay.OriginalFileObject
77020|                        );
77021|
77022|         SbInitOtherIrpStack(
77023|
    | NewIrpCurrentStack,
77024|
    | WriteRequest->DeviceObject,
77025|
    | currentIrpStack->FileObject,
77026|
    | currentIrpStack->Flags,
77027|                        WriteRequest,
77028|                        NULL,
77029|                        NULL,
77030|                        NULL
77031|                );
77032|
77033|                // Set up the next I/O stack
    | location. These are the parameters
77034|                // that will be passed to the
    | underlying driver.
77035|                nextIrpStack =
    | IoGetNextIrpStackLocation(NewIrp);
77036|
77037|                // make sure to not read past end
    | of volume, or the underlying driver (usually
    | ftdisk.sys)
77038|                // will fail the request with
    | STATUS_INVALID_PARAMETER
77039|                // we will do this by issuing the
    | read for less data than a granule, so the data
77040|                // after the requested area will be
    | junk
77041|                LARGE_INTEGER AdjustedCount =
    | WriteRequest->RoundedCountInBytes;
77042|

```

```

77043|
| if(WriteRequest->RoundedSectorInBytes.QuadPart+AdjustedC
| ount.QuadPart>DevExt->Pi.PartitionLength.QuadPart) {
77044|         Debug(DEBUG_THREAD,("Writer:
| Request for write past end of drive, adjusting\n"));
77045|         AdjustedCount.QuadPart =
| DevExt->Pi.PartitionLength.QuadPart-WriteRequest->RoundedSectorInBytes.QuadPart;
77046|     }
77047|
77048|     ASSERT(AdjustedCount.LowPart>=WriteRequest->ByteLength);
| //make sure this request isnt smaller than the passed
| in request
77049|     ASSERT(AdjustedCount.HighPart==0);
| //make sure the I/O length fits in 32 bits
77050|
77051|     SblInitReadIrpStack(
77052|         nextIrpStack,
77053|         | DevExt->TargetDeviceObject,
77054|         | currentIrpStack->FileObject,
77055|         | currentIrpStack->Flags,
77056|         | &WriteRequest->RoundedSectorInBytes,
77057|         | AdjustedCount.LowPart,
77058|         | currentIrpStack->Parameters.Write.Key
77059|         );
77060|
77061|         // We use a completion routine to
| keep the I/O Manager from doing
77062|         // "cleanup" on our IRP - like
| freeing our MDL.
77063|
77064|         // Pass Original Irp as context for
| function.
77065|         /*lint -save -e506 -e774 */
77066|         IoSetCompletionRoutine(NewIrp,
| PSMANWriteCompletionDevice, WriteRequest->Irp, TRUE,
| TRUE, TRUE);
77067|         /*lint -restore */
77068|
77069|         //Debug(DEBUG_THREAD |
| DEBUG_INFO,("Calling driver (tdo=%p, NewIrp=%p, Do=%p,
| Irp=%p)\n",DevExt->TargetDeviceObject,NewIrp,DeviceObject,
| t,Irp));

```

```

77070|          // send to caller
77071|
77072|          TRACE(
77073|              TRACE_SENDINGREADFORWRITE,
77074|              0,
77075|
77076|          | WriteRequest->RealSector.HighPart,
77077|          | WriteRequest->RealSector.LowPart,
77078|          | WriteRequest->RealCount,
77079|          | "");
77080|          //Debug(DEBUG_THREAD,("Writer:
77081|          | Sending read to lower driver,
77082|          | irql=%d\n",KeGetCurrentIrql()));
77083|
77084|          #if 1
77085|              (void)IoCallDriver(
77086|              | DevExt->TargetDeviceObject,NewIrp );
77087|          #else
77088|              PSMANWriteCompletionDevice(
77089|              | DeviceObject, NewIrp, Irp );
77090|          #endif
77091|          } else {
77092|              goto TimeToExit;
77093|          }
77094|          } else {
77095|              TimeToExit:
77096|              // most likely STATUS_WAIT_0 but could
77097|              | be an error condition
77098|              // or PSM_ERROR_DEADLOCK
77099|              //Debug(DEBUG_THREAD,("Writer: Error!
77100|              | %08x on wait\n",Status));
77101|              ExitThread = 1;
77102|
77103|              // if any outstanding writes, send them
77104|              | down so we can exit
77105|              // gracefully
77106|              SbCompleteWritesOnQueue();
77107|          }
77108|          }
77109|          } __except(
77110|          | ExceptionFilter(GetExceptionInformation()) ) {
77111|              Status=GetExceptionCode();
77112|              Debug(DEBUG_THREAD,("Writer: Error! Exception
77113|              | %08x\n",Status));
77114|              FailRequest(NULL,Status);
77115|          }

```

```

77109|
77110| #ifndef SYNC
77111|   ExitThreadJmp:
77112| #endif
77113|
77114|   TRACE( TRACE_THREADEXIT, 0, 0, 0, 0, "" );
77115|
77116|   Debug(DEBUG_THREAD,("Writer: Waiting for other
| threads to finish\n"));
77117|   while ( GlobalThreadCount>1 ) {
77118|       LARGE_INTEGER TimeToWait;
77119|
77120|       TimeToWait.QuadPart = RELATIVE(SECONDS(1));
77121|
77122|       KeDelayExecutionThread(
| (KPROCESSOR_MODE)KernelMode, FALSE, &TimeToWait );
77123|   }
77124|   SbCompleteWritesOnQueue();
77125|
77126|   SbDecrementRunningThreads();
77127|
77128|   Debug(DEBUG_THREAD,("Writer: Exiting\n"));
77129|   PsTerminateSystemThread( 0 );
77130| }
77131|
77132| #ifdef DEBUG
77133| /*-----
| -----*/
77134| NTSTATUS
77135| PSManIoSendWriteCompletionDevice(
77136|     IN PDEVICE_OBJECT
| DeviceObject,
77137|     IN PIRP      Irp,
77138|     IN PVOID
| Context
77139|     )
77140|
77141| /*++
77142|
77143| Routine Description:
77144|
77145|   This routine will get control from the system at
| the completion of an IRP.
77146|   It will calculate the difference between the time
| the IRP was started
77147|   and the current time, and decrement the queue
| depth.
77148|
77149|   IRQL <= DISPATCH_LEVEL Assume running at
| DISPATCH_LEVEL!!!

```

```

77150|     as it will vary depending on what the higher
77151|     | level driver is doing
77152| Arguments:
77153|
77154|     DeviceObject - for the IRP.
77155|     Irp          - The I/O request that just completed.
77156|     Context      - Not used.
77157|
77158| Return Value:
77159|
77160|     The IRP status.
77161|
77162| --*/
77163|
77164| {
77165|     PIO_STACK_LOCATION irpStack      =
77166|         | IoGetCurrentIrpStackLocation(Irp);
77167|     NOT_REFERENCED(Context);
77168|
77169|     TRACE( TRACE_SENDINGORIGWRITECOMP,
77170|         0,
77171|         | (long)(irpStack->Parameters.Read.ByteOffset.QuadPart /
77172|         | 512),
77173|         irpStack->Parameters.Read.Length / 512,
77174|         irpStack->Parameters.Read.Key,
77175|         "");
77176|     InterlockedDecrement( (PLONG)&OutstandingRequests
77177|         | );
77178|     if ( Irp->PendingReturned ) {
77179|         IoMarkIrpPending(Irp);
77180|     }
77181|
77182|     return STATUS_SUCCESS;
77183|
77184| } // PSMANIoSendWriteCompletionDevice
77185|
77186| /*-----
77187| | -----*/
77188| NTSTATUS
77189| PSMANSendOrigWrite(
77190|     IN PDEVICE_OBJECT DeviceObject,
77191|     IN PIRP Irp
77192| )
77193| /*++

```

```

77194|
77195| Routine Description:
77196|
77197|   This is the driver entry point for read requests
77198|   to disks to which the PSMAN driver has attached.
77199|   This driver collects statistics and then sets a
       | completion
77200|   routine so that it can collect additional
       | information when
77201|   the request completes. Then it calls the next
       | driver below
77202|   it.
77203|
77204| Arguments:
77205|
77206|   DeviceObject
77207|   Irp
77208|
77209| Return Value:
77210|
77211|   NTSTATUS
77212|
77213| --*/
77214|
77215| {
77216|   PFILTERED_EXTENSION deviceExtension =
       | GetFilteredExtension(DeviceObject);
77217| //   PIO_STACK_LOCATION currentIrpStack =
       | IoGetCurrentIrpStackLocation(Irp);
77218| //   PIO_STACK_LOCATION nextIrpStack =
       | IoGetNextIrpStackLocation(Irp);
77219|   NTSTATUS Status;
77220|
77221|   InterlockedIncrement( (PLONG)&OutstandingRequests
       | );
77222|
77223|   // Copy current stack to next stack.
77224|   IoCopyCurrentIrpStackLocationToNext(Irp);
77225|
77226|   // Set completion routine callback.
77227|   /*lint -save -e506 -e774 */
77228|   IoSetCompletionRoutine(Irp,
77229|       | PSMANIoSendWriteCompletionDevice,
77230|       DeviceObject,
77231|       TRUE,
77232|       TRUE,
77233|       TRUE);
77234|   /*lint -restore */
77235|

```



```

77236|    // Return the results of the call to the disk
      | driver.
77237|
77238|    Status =
      | IoCallDriver(deviceExtension->TargetDeviceObject, Irp);
77239|
77240|    return Status;
77241|
77242| } // end PSMANSendOrigWrite()
77243| #endif
77244|
77245| /*
77246|    You can not call IoCallDriver at >= DISPATCH_LEVEL
      | and completion routines can be
77247|    called back at <=DISPATCH_LEVEL, so we use this
      | thread to complete the writes
77248| */
77249| void SendWriteAfterReadThread( PVOID Context )
77250| {
77251|     NTSTATUS      Status={0};
77252|     ULONG         ExitThread=0;
77253|     PLIST_ENTRY    ListEntry=NULL;
77254|     PDEVICE_OBJECT DeviceObject=NULL;
77255|     PIRP           Irp=NULL;
77256|     tWriteRequest  *WriteRequest=NULL;
77257|     PIO_STACK_LOCATION IrpStack=NULL;
77258|
77259|     NOT_REFERENCED(Context);
77260|
77261|     PAGED_CODE();
77262|
77263|     pmAcquireMutex ( &VdiskThreadMutex, NULL);
77264|     VdiskNumberOfThreads++;
77265|     pmReleaseMutex ( &VdiskThreadMutex);
77266|
77267|     Debug(DEBUG_THREAD,("WriteAfterRead: Started
      | T=%08x, P=%08x,
      | Irql=%d\n",KeGetCurrentThread(),IoGetCurrentProcess(),Ke
      | GetCurrentIrql()));
77268|
77269|     // threads normally start with variable priority.
      | This sets it to the
77270|     // lowest of the time critical priorities
77271|     // Realtime threads have no quantum timeout. They
      | only give up the
77272|     // cpu when they voluntarily go into a wait state,
      | or when preempted by
77273|     // a thread of higher priority
77274|     //KeSetPriorityThread( KeGetCurrentThread(),
      | LOW_REALTIME_PRIORITY );

```

```

77275|
77276|     __try {
77277|         while ( !ExitThread ) {
77278|
77279|             Status = SbWaitForWriteAfterRead();
77280|
77281|             if ( Status == STATUS_WAIT_1 ) {
77282|
77283|                 ListEntry = ExInterlockedRemoveHeadList
77284|                     | (
77285|                         | &WriteAfterReadQueue,    // List Head
77286|                         | &WriteAfterReadSpinLock    // Lock
77287|                     | );
77288|                 if ( (ListEntry) &&
77289|                     | (ListEntry!=&WriteAfterReadQueue) ) {
77290|                     /*lint -save -e413 */
77291|                     WriteRequest = CONTAINING_RECORD(
77292|                         | ListEntry, tWriteRequest, ListEntry );
77293|                     /*lint -restore */
77294|                     DeviceObject =
77295|                         | WriteRequest->DeviceObject;
77296|                     Irp          = WriteRequest->Irp;
77297|                     FREE_POINTER(WriteRequest);
77298|
77299| #ifdef DEBUG
77300|                     IrpStack =
77301|                         | IoGetCurrentIrpStackLocation( Irp );
77302|                         // send write down
77303|                     TRACE( TRACE_SENDINGORIGWRITE, 0,
77304|                         | (long)(IrpStack->Parameters.Write.ByteOffset.QuadPart /
77305|                         | 512) , IrpStack->Parameters.Write.Length / 512, 0, "" );
77306| #endif
77307|                     InterlockedDecrement(
77308|                         | (PLONG)&OutstandingRequests );
77309| #ifdef DEBUG
77310|                     // so we know when the io completed
77311|                     (void)PSManSendOrigWrite(
77312|                         | DeviceObject, Irp);
77313| #else
77314|                     (void)PSManPassThru( DeviceObject,
77315|                         | Irp);
77316| #endif
77317|                 } else {
77318|                     | Debug(DEBUG_THREAD,("WriteAfterRead: Error ListEntry is

```

```

    | empty\n"));
77311|         }
77312|     } else {
77313|         // most likely STATUS_WAIT_0 but could
    | be an error condition
77314|         // or PSM_ERROR_DEADLOCK
77315|         //Debug(DEBUG_THREAD,("WriteAfterRead:
    | Error! %08x on wait\n",Status));
77316|         ExitThread = 1;
77317|     }
77318| }
77319| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
77320|     Status=GetExceptionCode();
77321|     Debug(DEBUG_THREAD,("WriteAfterRead: Error!
    | Exception %08x\n",Status));
77322|     FailRequest(NULL,Status);
77323| }
77324|
77325|
77326| // if any outstanding writes, send them down so we
    | can exit
77327| // gracefully
77328| SbCompleteWritesReadQueue();
77329|
77330| Debug(DEBUG_THREAD,("WriteAfterRead: Exiting\n"));
77331|
77332| pmAcquireMutex ( &VDiskThreadMutex, NULL);
77333| VDiskNumberOfThreads--;
77334| pmReleaseMutex ( &VDiskThreadMutex);
77335|
77336| PsTerminateSystemThread( 0 );
77337| }
77338|
77339|
77340| /*-----
    | -----*/
77341|
77342| NTSTATUS
77343| PSMANWriteCompletionDevice(
77344|     IN PDEVICE_OBJECT
    | DeviceObject,
77345|     IN PIRP      Irp,
77346|     IN PVOID      Context
77347| )
77348|
77349| /*++
77350|
77351| Routine Description:
77352|

```

```

77353| This routine will get control from the system at
| the completion of the
77354| read IRP we sent down instead of the read. It will
| queue the data to
77355| writer thread, and send down the original write.
77356|
77357| IRQL <= DISPATCH_LEVEL Assume running at
| DISPATCH_LEVEL!!!
77358| as it will vary depending on what the higher
| level driver is doing
77359|
77360| Arguments:
77361|
77362| DeviceObject - for the IRP.
77363| Irp - The I/O request that just completed.
77364| Context - Not used.
77365|
77366| Return Value:
77367|
77368| The IRP status.
77369|
77370| --*/
77371|
77372| {
77373|
77374| PIO_STACK_LOCATION NewIrpStackLoc =
| IoGetCurrentIrpStackLocation( Irp );
77375| PIRP OrigIrp = (PIRP)Context;
77376| PFILTERED_EXTENSION DevExt =
| GetFilteredExtension(DeviceObject);
77377| PIO_STACK_LOCATION irpStack =
| IoGetCurrentIrpStackLocation(OrigIrp);
77378| NTSTATUS Status;
77379| KIRQL oldIrql;
77380| tWriteRequest *WriteRequest;
77381|
77382| ASSERT(DevExt->ObjectType==OBJECT_FILTEREDDISK);
77383| //Debug(DEBUG_THREAD |
| DEBUG_PROCCALL,("PSManWriteCompletionDevice Called
| device = %p irp = %p OrigIrp =
| %p\n",DeviceObject,Irp,Context));
77384|
77385| //Debug(DEBUG_THREAD,("ReadForWrite Done:
| DeviceObject=%p, Irp=%p,
| OrigIrp=%p\n",DeviceObject,Irp,OrigIrp));
77386| // arg1 = WriteRequest
77387| ASSERT(Irp->AssociatedIrp.SystemBuffer == NULL);
77388| WriteRequest =
| (tWriteRequest*)NewIrpStackLoc->Parameters.Others.Argume
| nt1;

```

```

77389|
77390|  TRACE(
77391|      TRACE_READFORWRITE_COMP,
77392|      0,
77393|      WriteRequest->RealSector.HighPart,
77394|      WriteRequest->RealSector.LowPart,
77395|      WriteRequest->RealCount,
77396|      "");
77397|
77398| #ifdef DEBUG
77399|  if ( DebugPrints ) {
77400|      File_PrintOneLiner("Write Done:
77401|  | ",DeviceObject,OrigIrp);
77402|  }
77403| #endif
77404|  if ( Irp->PendingReturned ) {
77405|      //Debug(DEBUG_THREAD |
77406|  | DEBUG_INFO,("ReadForWrite: Marking device=%p, Irp=%p
77407|  | pending\n",DeviceObject,Irp));
77408|      IoMarkIrpPending(Irp);
77409|  }
77410|  // unlock the pages and free the Mdl, as we no
77411|  | longer need page locked
77412|  // memory...
77413|  if ( Irp->MdlAddress ) {
77414|      //Debug(DEBUG_THREAD,("Freeing Mdl
77415|  | %p\n",Irp->MdlAddress));
77416|      MmUnlockPages(Irp->MdlAddress);
77417|      IoFreeMdl(Irp->MdlAddress);
77418|  }
77419|  // can not access Irp after we have queued it
77420|  Status = Irp->IoStatus.Status;
77421|  if ( NT_SUCCESS(Status) ) {
77422|      // do statistics
77423|      pmAcquireSpinLock (
77424|  | &DevExt->StatisticsSpinLock, &oldIrql );
77425|      // update the logical partition
77426|      DevExt->SectorsWritten +=
77427|  | WriteRequest->ByteLength;
77428|      DevExt->NumberOfWriteRequests++;
77429|      // update the physical drive
77430|      //PhyExt->SectorsWritten +=
77431|  | (irpStack->Parameters.Write.Length /
77432|  | DevExt->BytesPerSector );
77433|      //PhyExt->NumberOfWriteRequests++;

```

```

77430|     pmReleaseSpinLock (
    | &DevExt->StatisticsSpinLock, oldIrql );
77431| } else {
77432|     // hmmm the read failed for some reason. lets
    | fail the write
77433|     // since we could not protect the data at this
    | sector (assuming it has data)
77434|     OrigIrp->IoStatus.Status =
    | Irp->IoStatus.Status;
77435|     OrigIrp->IoStatus.Information =
    | Irp->IoStatus.Information;
77436|
77437|     Debug(DEBUG_THREAD,("Read for write Failed,
    | Irp=%08x, Status = %08x,
    | OrigIrp=%08x\n",Irp,Status,OrigIrp));
77438|
77439| #if 0
77440|     File_PrintOneLiner("Read Failed:
    | ",DeviceObject,Irp);
77441|     File_PrintOneLiner(" for write
    | ",DeviceObject,OrigIrp);
77442|     File_PrintIrp("Read Failed:
    | ",DeviceObject,Irp);
77443|     File_PrintIrp(" for write
    | ",DeviceObject,OrigIrp);
77444|     DbgBreakPoint();
77445| #endif
77446| #ifdef DEBUG
77447|     switch(Status) {
77448|         case STATUS_DEVICE_BUSY :
77449|         case STATUS_DEVICE_OFF_LINE :
77450|             // dont enter the debugger on these
    | errors
77451|             break;
77452|         case STATUS_INSUFFICIENT_RESOURCES :
77453|         default:
77454|             DbgBreakPoint();
77455|     }
77456| #endif
77457|     InterlockedDecrement(
    | (PLONG)&OutstandingRequests );
77458|
77459|     // complete with DISK increment since we did do
    | IO.
77460|     IoCompleteRequest(OrigIrp, IO_DISK_INCREMENT);
77461| }
77462|
77463| //Debug(DEBUG_THREAD,("Writer: Adding %p to
    | queue\n",Irp));
77464| // add to queue...

```

```

77465|   ExInterlockedInsertTailList (
       | &ThreadsWorkToDoQueue,
77466|
       | &Irp->Tail.Overlay.ListEntry,
77467|
       | &ThreadsWorkToDoSpinLock);
77468|
77469|   //Debug(DEBUG_THREAD,("Writer: Semaphore Count =
       | %d\n",pmExamineSemaphore(&ThreadSemaphore)));
77470|   // have a thread work on it.
77471|   pmSignalSemaphore( &ThreadSemaphore);
77472|
77473| //   Debug(DEBUG_THREAD |
       | DEBUG_PROCCALL,("PSManWriteCompletionDevice Done\n"));
77474|   return STATUS_MORE_PROCESSING_REQUIRED;
77475|
77476| } // PSManWriteCompletionDevice
77477|
77478|
77479|
77480| File Listing: THREAD.h
77481|
77482| STATIC void SbIncrementRunningThreads( void );
77483| STATIC void SbDecrementRunningThreads( void );
77484| STATIC int SbThreadQueueUpdate ( int Up );
77485| STATIC int SbThreadQueueInc (void);
77486| STATIC int SbThreadQueueDec (void);
77487| STATIC NTSTATUS SbThreadQueueWait (void);
77488| STATIC NTSTATUS SbWaitForThreadWork(void);
77489| STATIC NTSTATUS SbWaitOnEvent ( PKEVENT Event );
77490| STATIC NTSTATUS SbThreadInit (
77491|             ULONG ThreadNum
77492|             );
77493| PIRP SbGetWorkItem (void);
77494| STATIC PIRP SbGetWork(
77495|             PDEVICE_OBJECT
       | *DeviceObject,
77496|             PIO_STACK_LOCATION *Stack,
77497|             PFILTERED_EXTENSION
       | *DeviceExtension,
77498|             ULONG          *Sector,
77499|             ULONG          *Count,
77500|             PCHAR          *Buffer
77501|             );
77502|
77503| STATIC int SbMakeAnotherThread ( void );
77504| STATIC NTSTATUS SbWaitForWriteFromNT(void);
77505| STATIC NTSTATUS SbCompleteNextWriteOnQueue(void);
77506| NTSTATUS SbCompleteWritesOnQueue(void);
77507| STATIC NTSTATUS SbWaitForFreeThread(void);

```

```

77508| STATIC NTSTATUS SbAllocateIrpResources ( PCHAR *Buffer,
77509|                                     PIRP *Irp,
77510|                                     PMDL *Mdl,
77511|                                     USHORT
    | StackSize,
77512|                                     ULONG ByteCount
77513|                                     );
77514| STATIC NTSTATUS SbAllocateIrpResourcesOrWait ( PCHAR
    | *Buffer,
77515|                                     PIRP *Irp,
77516|                                     PMDL *Mdl,
77517|                                     USHORT
    | StackSize,
77518|                                     ULONG
    | ByteCount
77519|                                     );
77520| STATIC NTSTATUS SbInitReadIrp(
77521|     PIRP     NewIrp,
77522|     PMDL     Mdl,
77523|     PETHREAD Thread,
77524|     PCHAR     AuxiliaryBuffer,
77525|     PFILE_OBJECT
    | OriginalFileObject
77526|     );
77527| STATIC NTSTATUS SbInitOtherIrpStack (
77528|     PIO_STACK_LOCATION
    | Stack,
77529|     PDEVICE_OBJECT
    | DeviceObject,
77530|     PFILE_OBJECT
    | FileObject,
77531|     UCHAR
    | Flags,
77532|     PVOID
    | Arg1,
77533|     PVOID
    | Arg2,
77534|     PVOID
    | Arg3,
77535|     PVOID
    | Arg4
77536|     );
77537| STATIC NTSTATUS SbInitReadIrpStack(
77538|     PIO_STACK_LOCATION
    | Stack,
77539|     PDEVICE_OBJECT
    | DeviceObject,
77540|     PFILE_OBJECT
    | FileObject,
77541|     UCHAR

```



```

    | Flags,
77542|                PLARGE_INTEGER
    | ByteOffset,
77543|                ULONG
    | Length,
77544|                ULONG
    | Key
77545|                );
77546|
77547|
77548|
77549|
77550| void SaveOriginalDataThread( PVOID Context );
77551| void WriteDispatchThread ( PVOID Context );
77552| void SendWriteAfterReadThread( PVOID Context );
77553|
77554| void CacheThresholdReached(PDEVICE_OBJECT Volume);
77555|
77556| NTSTATUS PSManWriteCompletionDevice(
77557|     IN PDEVICE_OBJECT DeviceObject,
77558|     IN PIRP         Irp,
77559|     IN PVOID         Context
77560| );
77561|
77562| //-----
    | -----
77563| // DeleteOldestSnapShot
77564| //
77565| // Returns TRUE if a snapshot was deleted, FALSE
    | otherwise.
77566| //
77567| BOOLEAN DeleteOldestSnapShot(PDEVICE_OBJECT Volume,
    | BOOLEAN DeleteAlwaysKeep );
77568|
77569|
77570|
77571| File Listing: UNLOAD.cpp
77572|
77573| #include "precomp.h"
77574|
77575| VOID
77576| PSManUnload(
77577|     IN PDRIVER_OBJECT DriverObject
77578| )
77579|
77580| {
77581|     PSBPSMAN_EXTENSION DevExt=
    | (PSBPSMAN_EXTENSION)GetDeviceExtension(PSManObject);
77582|     WCHAR         deviceLinkBuffer[100]={0};
77583|     UNICODE_STRING deviceLinkUnicodeString={0};

```

```

77584|
77585|   Debug(DEBUG_INIT,("Psm: Unloading...\n"));
77586|
77587|   ExDeleteResourceLite(&DevExt->DeviceResource);
77588|   pmDeRegisterObject(&DevExt->DeviceResource);
77589|   ExDeleteResourceLite(&DevExt->SnapShotResource);
77590|   pmDeRegisterObject(&DevExt->SnapShotResource);
77591|
77592|   ZwClose(gVDiskRootDirHandle);
77593|
77594|   // Create a symbolic link, e.g. a name that a Win32
       | app can specify
77595|   // to open the device
77596|
77597|   | swprintf(deviceLinkBuffer,L"%s_%04x",SBPSMAN_WIN32_NAME,
       | PSM_LOW_COMPATIBLE_VERSION);
77598|   RtlInitUnicodeString(&deviceLinkUnicodeString,
       | deviceLinkBuffer);
77599|   IoDeleteSymbolicLink (&deviceLinkUnicodeString );
77600|
77601|   IoDeleteDevice(PsManObject);
77602|   Debug(DEBUG_INIT,("Psm: Done Unloading...\n"));
77603|   return;
77604| }
77605|
77606|
77607|
77608| File Listing: UNLOAD.h
77609|
77610| VOID
77611| PsManUnload(
77612|   IN PDRIVER_OBJECT DriverObject
77613|   );
77614|
77615|
77616|
77617| File Listing: VDISK.cpp
77618|
77619| #include "precomp.h"
77620| #include <initguid.h>
77621|
77622| DEFINE_GUID(MOUNTDEV_MOUNTED_DEVICE_GUID, 0x53f5630d,
       | 0xb6bf, 0x11d0, 0x94, 0xf2, 0x00, 0xa0, 0xc9, 0x1e,
       | 0xfb, 0x8b);
77623|
77624| // Device Characteristics
77625|         // FILE_REMOVABLE_MEDIA
77626|         // FILE_READ_ONLY_DEVICE
77627|         // FILE_FLOPPY_DISKETTE

```

```

77628|          // FILE_WRITE_ONCE_MEDIA
77629|          // FILE_REMOTE_DEVICE
77630|          // FILE_DEVICE_IS_MOUNTED
77631|          // FILE_VIRTUAL_VOLUME
77632|
77633| #define TDROM_CHARS (FILE_REMOVABLE_MEDIA |
    | FILE_VIRTUAL_VOLUME)
77634|
77635| // Device Types
77636|          // FILE_DEVICE_DISK
77637|          // FILE_DEVICE_VIRTUAL_DISK
77638|
77639| #define TDROM_TYPE (FILE_DEVICE_DISK)
77640|
77641| NTSTATUS VDisk_DismountDirtyVolume( PDEVICE_OBJECT
    | Volume );
77642|
77643| // unique value for each entry
77644| volatile ULONG VDiskInstance=0;
77645|
77646| /*
77647|    Given a directory name returns back a name with no
    | slashes
77648|    ie: "\\Device\\Harddisk0\\Partition1" would return
77649|        "_Device_Harddisk0_Partition1"
77650| */
77651| void NormalizeDeviceName( WCHAR *Name )
77652| {
77653|     ULONG Len=wcslen(Name);
77654|     ULONG i;
77655|     for(i=0;i<Len;i++) {
77656|         if(Name[i]==L'\\') {
77657|             Name[i] = L'_';
77658|         }
77659|     }
77660|     return;
77661| }
77662|
77663| /*
77664|    If any other snapshots are using this instance then
    | it is not safe
77665| */
77666| BOOLEAN SafeToUseInstance( tkSnapShotMaster
    | *MasterSnapShot, ULONG Instance )
77667| {
77668|     PDEVICE_OBJECT DevObj =
    | PSMAN_DRIVER_OBJECT->DeviceObject;
77669|     BOOLEAN Safe = TRUE;
77670|
77671|     while(DevObj) {

```

```

77672|
77673|     | if(PsmGetObjectype(DevObj)==OBJECT_VIRTUALDISK) {
77674|         PVDISK_EXTENSION
77675|         | DevExt=(PVDISK_EXTENSION)GetDeviceExtension(DevObj);
77676|
77677|         if( (MasterSnapShot !=
77678|             | DevExt->MasterSnapShot) &&
77679|             (DevExt->Instance == Instance) &&
77680|             (DevExt->PartitionActive)) {
77681|                 Safe = FALSE;
77682|             }
77683|         }
77684|         DevObj = DevObj->NextDevice;
77685|     }
77686|     return Safe;
77687| }
77688| /*-----*/
77689| /*
77690| Return an unused device object or create one if
77691| none exist
77692| This routine is NOT reentrant.
77693| */
77694| STATIC NTSTATUS GetADeviceObject ( PDRIVER_OBJECT
77695|     | DriverObject,
77696|     WCHAR *NameToUse,
77697|     PDEVICE_OBJECT
77698|     | *DeviceObject,
77699|     tkSnapShotMaster
77700|     | *MasterSnapShot,
77701|     CHAR
77702|     | AssignedInstance
77703| )
77704| {
77705|     PDEVICE_OBJECT DevObj = DriverObject->DeviceObject;
77706|     PVDISK_EXTENSION DevExt=NULL;
77707|     WCHAR Name[256]={0};
77708|     UNICODE_STRING NameUnicode={0};
77709|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
77710|
77711|     // search for the lowest number, this is so we
77712|     // do not start our number scheme going
77713|     | downwards
77714|     // this is only for user display and not any
77715|     | functional
77716|     // reason
77717|

```

```

77711| #if 1
77712|     ULONG Lowest=0xffffffff;
77713|     PDEVICE_OBJECT LowestDevObj=NULL;
77714|
77715|     while(DevObj) {
77716|
77717|         | if(PsmGetObjectTypes(DevObj)==OBJECT_VIRTUALDISK) {
77718|             DevExt =
77719|             | (PVDISK_EXTENSION)GetDeviceExtension(DevObj);
77720|
77721|             if( (_wcsicmp(NameToUse,DevExt->Name)==0)
77722|             | &&
77723|             | (!DevExt->PartitionActive) ) {
77724|                 if(AssignedInstance) {
77725|                     if
77726|                     | (DevExt->Instance==MasterSnapShot->Instance) {
77727|                         // use the passed in
77728|                         | instance number instead of looking for one
77729|                         Lowest
77730|                         | = DevExt->Instance;
77731|
77732|                         | LowestDevObj = DevObj;
77733|
77734|                         // we
77735|                         | found it so exit
77736|                         break;
77737|                     }
77738|                 } else
77739|                 | if(SafeToUseInstance(MasterSnapShot,DevExt->Instance))
77740|                 | {
77741|                     // dont use temporary
77742|                     | snapshots
77743|
77744|                     | if((DevExt->Instance<MAX_NUMBER_OF_SNAPSHOTS) &&
77745|                     | (DevExt->Instance<Lowest)) {
77746|                         Lowest =
77747|                         | DevExt->Instance;
77748|
77749|                         LowestDevObj =
77750|                         | DevObj;
77751|                     }
77752|                 }
77753|             }
77754|         }
77755|
77756|         DevObj = DevObj->NextDevice;
77757|     }
77758|
77759|     if(LowestDevObj) {
77760|         DevExt =

```

```

    | (PVDISK_EXTENSION)GetDeviceExtension(LowestDevObj);
77746|     (*DeviceObject) = LowestDevObj;
77747|
77748|     Debug(DEBUG_VDISK,("Init: Reusing Virtual disk
    | Device %p '%S':%d instead of
    | %d\n",LowestDevObj,NameToUse,DevExt->Instance,MasterSnap
    | Shot->Instance));
77749|
77750|     DevExt->DriveNotReady = TRUE;
77751|     DevExt->PartitionActive = TRUE;
77752|     DevExt->DeviceShutDown = 0;
77753|     return STATUS_SUCCESS;
77754| }
77755|
77756| #else
77757| while(DevObj) {
77758|
    | if(PsmGetObjectTypes(DevObj)==OBJECT_VIRTUALDISK) {
77759|     DevExt =
    | (PVDISK_EXTENSION)GetDeviceExtension(DevObj);
77760|
77761|     if( (_wcsicmp(NameToUse,DevExt->Name)==0)
    | &&
77762|     (!DevExt->PartitionActive) &&
77763|     |(SafeToUseInstance(MasterSnapShot,DevExt->Instance))
    | ) {
77764|         (*DeviceObject) = DevObj;
77765|
77766|         Debug(DEBUG_VDISK,("Init: Reusing
    | Virtual disk Device %p '%S':%d instead of
    | %d\n",DevObj,NameToUse,DevExt->Instance,MasterSnapShot->
    | Instance));
77768|
77769|         DevExt->DriveNotReady = TRUE;
77770|         DevExt->PartitionActive = TRUE;
77771|         return STATUS_SUCCESS;
77772|     }
77773| }
77774|
77775| DevObj = DevObj->NextDevice;
77776| }
77777| #endif
77778|
77779| // no device was found, so create one.
77780|
77781|
    | swprintf(Name,L"\\Device\\PsmDevices_%04x\\%s_%d",PSM_LO
    | W_COMPATIBLE_VERSION,NameToUse,MasterSnapShot->Instance
    | );

```

```

77782|
77783|     RtlInitUnicodeString( &NameUnicode, Name);
77784|
77785|     {
77786| #ifdef DEBUG_EXTENSION
77787|         ULONG SizeOfDeviceExt =
77788|             | sizeof(DEVICE_EXTENSION);
77789| #else
77790|         ULONG SizeOfDeviceExt =
77791|             | sizeof(VDISK_EXTENSION);
77792| #endif
77793|         Status = IoCreateDevice(
77794|             DriverObject,
77795|             SizeOfDeviceExt,
77796|             &NameUnicode,
77797|             TDROM_TYPE,
77798|             TDROM_CHARS,
77799|             FALSE,
77800|             &DevObj);
77801|     }
77802|     // enter the debugger
77803|     ASSERT(NT_SUCCESS( Status ));
77804|
77805|     if ( NT_SUCCESS( Status ) ) {
77806|         (*DeviceObject) = DevObj;
77807|
77808|         Debug(DEBUG_VDISK,("Init: Virtual disk Device
77809|             | %p '%S':%d created for driver
77810|             | %p\n",DevObj,NameToUse,MasterSnapShot->Instance,DriverOb
77811|             | ject));
77812|         // statisfy lint...
77813|         if(DevObj) {
77814| #ifdef DEBUG_EXTENSION
77815|             | ((PDEVICE_EXTENSION)(DevObj->DeviceExtension))->ObjectTy
77816|             | pe = OBJECT_VIRTUALDISK;
77817|             | ((PDEVICE_EXTENSION)(DevObj->DeviceExtension))->RealDevi
77818|             | ceExtension =
77819|             | MemAllocatePoolWithTag(NonPagedPool,sizeof(VDISK_EXTENSI
77820|             | ON),DEVEXTTAG);
77821|             | RtlZeroMemory(((PDEVICE_EXTENSION)(DevObj->DeviceExtensi
77822|             | on))->RealDeviceExtension,sizeof(VDISK_EXTENSION));
77823| #endif
77824|             DevExt = (PVDISK_EXTENSION)
77825|             | GetDeviceExtension(DevObj);
77826|             DevExt->ObjectType =

```

```

| OBJECT_VIRTUALDISK;
77818|     DevExt->PartitionActive = TRUE;
77819|     DevExt->DriveNotReady  = TRUE;
77820|     DevExt->Instance      =
| MasterSnapShot->Instance;
77821|         DevExt->DeviceShutDown = 0;
77822|     wcsncpy(DevExt->Name,NameToUse);
77823| }
77824|
77825| } else {
77826|     Debug(DEBUG_VDISK,("Init: Error creating %p
| '%S':%d\n",DevObj,NameToUse,MasterSnapShot->Instance));
77827| }
77828|
77829| return Status;
77830| }
77831|
77832| #include <initguid.h>
77833| DEFINE_GUID(PSM_VDISK_GUID, 0x6b23c937, 0x106e, 0x4f94,
| 0x96, 0x7d, 0xe6, 0x5b, 0x7e, 0x5f, 0xad, 0x9);
77834|
77835|
77836| /*-----
| -----*/
77837| /*
77838| This function is not reentrant
77839| */
77840| NTSTATUS TdAddDrive ( PRTL_BITMAP *CachingBitMap,
| PDEVICE_OBJECT DeviceToPSM, tkSnapShotMaster
| *MasterSnapShot, PDEVICE_OBJECT *VirtualDeviceObject,
| CHAR AssignedInstance )
77841| {
77842|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
77843|     PDEVICE_OBJECT DeviceObject=NULL;
77844|     PVDISK_EXTENSION DevExt=NULL;
77845|     LARGE_INTEGER BO;
77846|     PFILTERED_EXTENSION PartExt=NULL;
77847|     WCHAR *Name=(WCHAR*)MemAllocateString(256);
77848|
77849|
| ASSERT(PsmGetObjectype(DeviceToPSM)==OBJECT_FILTEREDDIS
| K);
77850|
77851|     if(!Name) {
77852|         return STATUS_INSUFFICIENT_RESOURCES;
77853|     }
77854|
77855| #ifdef DEBUG
77856|     if(!IsSnapShotAcquiredForWrite()) {
77857|         Debug(DEBUG_DCPSM,("TdAddDrive: Snapshot

```



```

    | resource not acquired!\n"));
77858|     DbgBreakPoint();
77859| }
77860| #endif
77861|
77862| // get physical device extension so we can get the
    | device params
77863|     PartExt =
    | (PFILTERED_EXTENSION)GetDeviceExtension(DeviceToPSM);
77864|
77865|     ASSERT(PartExt->ObjectType==OBJECT_FILTEREDDISK);
77866|
77867|     wcscpy(Name,PartExt->Name);
77868|     NormalizeDeviceName(Name);
77869|
77870|     // get a device, reusing an object if available.
77871|     Status = GetADeviceObject( PSMAN_DRIVER_OBJECT, Name,
    | &DeviceObject,MasterSnapShot, AssignedInstance );
77872|
77873|     if(!NT_SUCCESS(Status)) {
77874|         Debug(DEBUG_VDISK,("VDisk: Devcon: Error %08x
    | getting device object\n",Status));
77875|     }
77876|     // satisfy lint
77877|     if(DeviceObject) {
77878|
77879|
77880|         // Initialize device object and extension.
77881|
77882|         DeviceObject->Flags |= DO_DIRECT_IO;
77883|         DeviceObject->AlignmentRequirement =
    | DeviceToPSM->AlignmentRequirement;
77884|
77885|         DevExt =
    | (PVDISK_EXTENSION)GetDeviceExtension(DeviceObject);
77886|         ASSERT(DevExt->ObjectType==OBJECT_VIRTUALDISK);
77887|
77888|         // Make this disk a Unique Number... This is
    | to cause the file system to
77889|         // discard any cached data for the old volume
    | (due to a forced dismount)
77890|         // this however will cause all volumes to go
    | through the full mount stage
77891|         // even if the volume was not forced shutdown
77892|
77893|         /*lint -save -e740 */
77894|         KeQueryTickCount( &BO );
77895|         /*lint -restore */
77896|         DevExt->SerialNumber = BO.LowPart;
77897|

```

```

77898|    DevExt->DriverObject = PSManDriverObject;
77899|    DevExt->DeviceObject = DeviceObject;
77900|
77901|    DevExt->PSMDevice = DeviceToPSM;
77902|    DevExt->IsPhysical = PartExt->IsPhysical;
77903|
77904|    DevExt->Cylinders = PartExt->Cylinders;
77905|    DevExt->Heads =
    | PartExt->TracksPerCylinder;
77906|    DevExt->SPT = PartExt->SectorsPerTrack;
77907|    DevExt->BPS = PartExt->BytesPerSector;
77908|    DevExt->MountDisabled = FALSE;
77909|
77910|    // set to TRUE if this volume can be write
    | protected.
77911|    DevExt->OriginalWriteProtected = FALSE;
77912|
77913|    DevExt->LockCount =
77914|        DevExt->DiskChangeCount =
77915|        DevExt->DiskChangeCount =
77916|        DevExt->LockCount =
77917|        DevExt->NumberOfReadRequests =
77918|        DevExt->SectorsRead =
77919|        DevExt->NumberOfWriteRequests =
77920|        DevExt->SectorsWritten =
77921|        DevExt->CacheWrites = 0;
77922|
77923|    // record what instance this device is in case
    | it changed
77924|    MasterSnapShot->Instance = DevExt->Instance;
77925|
77926|    DevExt->MasterSnapShot = MasterSnapShot;
77927|    DevExt->SnapShot =
    | FindSnapShotEntryForDevice(MasterSnapShot, DeviceToPSM);
77928|
77929|    UseSnapShot(DevExt->SnapShot);
77930|
77931|    // compute length of drive in bytes.
77932|    DevExt->Pi = PartExt->Pi;
77933|
77934|    DevExt->PartitionActive = TRUE;
77935|    DevExt->DriveNotReady = FALSE;
77936|
77937|    // Clear the device's init flag as per NT DDK
    | KB article on creating device
77938|    // objects from a dispatch routine
77939|    // From this point on, we will get Irp's
77940|    DeviceObject->Flags &= ~DO_DEVICE_INITIALIZING;
77941|
77942|    *VirtualDeviceObject = DeviceObject;

```

```

77943|
77944|     Debug(DEBUG_VDISK,("VDisK: Devcon: Using %p
| ('%S',%d) in master %08x
| %08x\n",DeviceObject,DevExt->Name,DevExt->Instance,Maste
| rSnapShot,DevExt->SnapShot));
77945|
77946|     UNICODE_STRING SymLink;
77947|     UNICODE_STRING Target;
77948|     WCHAR *Buffer=(WCHAR*)MemAllocateString(256);
77949|     WCHAR *Buffer2=(WCHAR*)MemAllocateString(256);
77950|     UUID Guid;
77951|
77952|     if(Buffer && Buffer2) {
77953|
| wcsncpy(Buffer,L"\\??\\Volume{00000000-0000-0000-0000-000
| 00000000}");
77954|
77955|         // get a guid
77956|         // while(ExUuidCreate(&Guid)!=0);
77957|         Guid = PSM_VDISK_GUID;
77958|         Guid.Data1 = PartExt->Volumeld;
77959|         if(Guid.Data1!=0) {
77960|             Guid.Data2 = (USHORT)DevExt->Instance;
77961|
77962|             RtlStringFromGUID(Guid,&Target);
77963|
| RtlCopyMemory(Buffer+10,Target.Buffer,Target.Length);
77964|
77965|             RtlFreeUnicodeString(&Target);
77966|
77967|             RtlInitUnicodeString(&SymLink, Buffer);
77968|
77969|             // save in devext
77970|             wcsncpy(DevExt->VolumeGuid,Buffer);
77971|
77972|
| swprintf(Buffer2,L"\\Device\\PsmDevices_%04x\\%s_%d",PSM
| _LOW_COMPATIBLE_VERSION,DevExt->Name,DevExt->Instance);
77973|             RtlInitUnicodeString(&Target,Buffer2 );
77974|
77975|             Status = IoCreateSymbolicLink
| (&SymLink,&Target);
77976|             if(NT_SUCCESS(Status)) {
77977|                 Debug(DEBUG_VDISK,("Success
| creating link '%S' to
| '%S'\n",SymLink.Buffer,Target.Buffer));
77978|
77979|                 ReleaseSnapShotForWrite();
77980|                 __try {
77981|                     EnableWritesToNewFiles();

```

```

77982|         if(gVDiskDoVirtualIO) {
77983|             __try {
77984|                 // make sure the volume
77985|                 | is mounted
77986|                 SbTouchVolume(Buffer2);
77987|                 // lets delete the
77988|                 | snapshot directory off of the virtual drive
77989|                 // so nesting will not
77990|                 | occur when other snapshots exist
77991|                 wscpy(Buffer,Buffer2);
77992|                 wscat(Buffer,L"\");
77993|                 | wscat(Buffer,gSnapShotDirName);
77994|                 | Debug(DEBUG_DEVSUP,("Deleting directory
77995|                 | '%S'\n",Buffer));
77996|                 | SbSnapShotCleanup(Buffer);
77997|                 } __finally {
77998|                 | if(AbnormalTermination()) {
77999|                 | DisableWritesToNewFiles();
78000|                 | }
78001|                 | }
78002|             #if 0
78003|                 // Incase the junction point
78004|                 | has changed since the last
78005|                 // time we mounted this volume.
78006|                 | This can happen on cluster
78007|                 // server when the volume fails
78008|                 | over from one system to another
78009|                 // if the user has already set
78010|                 | the name
78011|                 | if(MasterSnapShot->UserSnapShotName[0]!=0) {
78012|                 // make junction point
78013|                 swprintf (
78014|                 | Buffer,L"%s\\%s",PartExt->Name,MasterSnapShot->UserSnapS
78015|                 | hotName );
78016|                 swprintf ( Buffer2,
78017|                 | L"%s\\",DevExt->VolumeGuid );
78018|                 | Debug(DEBUG_VDISK,("Creating junction '%S' to
78019|                 | '%S'\n",Buffer,Buffer2));

```

```

78013|             CreateJunction
| (Buffer,Buffer2,MasterSnapShot->SnapShotTime);
78014|         } else {
78015|             Debug(DEBUG_VDISK,("Cant
| create junction as we dont know name yet\n"));
78016|         }
78017|     #endif
78018|
| if(PersistentDictionary::DoFreeSpaceChecks()) {
78019|         __try {
78020|             GetSnapShotForRead()
78021|         __try {
78022|             pDictionary Dict;
78023|
| PersistentDictionary::GetDictionaryForVolume(DeviceToPSM
| ,Dict);
78024|
78025|             if
| (((pPersistentDictionary)Dict)->ProcessCachingMap(
| CachingBitMap, Buffer2, PartExt->Name)!=STATUS_SUCCESS)
| {
78026|                 // FIXFIXFIX
| what to do???? - this could messup snapshots
78027|                 // - maybe
| switch off free_space ????
78028|
| Debug(DEBUG_VDISK,("Error adjusting volume bitmap\n"));
78029|             }
78030|         } __finally {
78031|
| ReleaseSnapShotForRead();
78032|         }
78033|     } __finally {
78034|
| if(AbnormalTermination()) {
78035|         DisableWritesToNewFiles();
78036|     }
78037| }
78038| } // DoFreeSpaceChecks
78039| DisableWritesToNewFiles();
78040| } __finally {
78041|     GetSnapShotForWrite();
78042| }
78043| } else {
78044|     Debug(DEBUG_VDISK,("Error %08x
| creating link '%S' to
| '%S'\n",Status,SymLink.Buffer,Target.Buffer));
78045| }
78046| } else { // guid!=0

```

```

78047|         Debug(DEBUG_VDISK,("Guid is 0\n"));
78048|     }
78049|
78050| #if 0
78051|         UNICODE_STRING SymLink={0};
78052|         // register the volume with the mount
        | manager
78053|         Status =
        | IoRegisterDeviceInterface(DeviceObject,&MOUNTDEV_MOUNTED
        | _DEVICE_GUID,NULL,&SymLink);
78054|         if(NT_SUCCESS(Status)) {
78055|             Debug(DEBUG_VDISK,("Vdisk:
        | Registered successful,
        | symlink='%S'\n",SymLink.Buffer));
78056|             RtlFreeUnicodeString(&SymLink);
78057|         } else {
78058|             Debug(DEBUG_VDISK,("Vdisk: register
        | failed with %08x\n",Status));
78059|         }
78060| #endif
78061|         MemFreeString(Buffer);
78062|         MemFreeString(Buffer2);
78063|         Status = STATUS_SUCCESS;
78064|     } else {
78065|         Status = STATUS_INSUFFICIENT_RESOURCES;
78066|         Debug(DEBUG_VDISK,("Vdisk: Out of memory
        | for volume names\n"));
78067|     }
78068| }
78069|
78070| MemFreeString(Name);
78071| return Status;
78072| }
78073|
78074| NTSTATUS TdDelVirtualDrive ( PDEVICE_OBJECT
        | VirtualDrive )
78075| {
78076|     __try {
78077|         PVDISK_EXTENSION DevExt = (PVDISK_EXTENSION)
        | GetDeviceExtension(VirtualDrive);
78078|
78079|         Debug(DEBUG_VDISK,("Vdisk: Devcon: Cleaning up
        | %p ('%S',%d) Reference=%d, Master= %08x
        | %08x\n",VirtualDrive,DevExt->Name,DevExt->Instance,Virtu
        | alDrive->ReferenceCount,DevExt->MasterSnapShot,DevExt->S
        | napShot));
78080|         Debug(DEBUG_VDISK,("Vdisk: Devcon:
        | ChangeCount=%d,
        | LockCount=%d\n",DevExt->DiskChangeCount,DevExt->LockCoun
        | t));

```

```

78081|
78082|     if ((DevExt->DeviceObject->Vpb) && (
78083|         | DevExt->DeviceObject->Vpb->Flags & VPB_MOUNTED )) {
78084|         Debug(DEBUG_VDISK,("VDisk: Devcon: Volume
78085|         | %08x is mounted\n",VirtualDrive));
78086|         DevExt->DeviceObject->Flags |=
78087|         | DO_VERIFY_VOLUME;
78088|     }
78089|
78090|     DevExt->PartitionActive = FALSE;
78091|     DevExt->DriveNotReady = TRUE;
78092|     DevExt->PSMDevice = NULL;
78093|     DevExt->MountDisabled = TRUE;
78094|
78095|     UNICODE_STRING Uni;
78096|
78097|     RtlInitUnicodeString(&Uni, DevExt->VolumeGuid);
78098|     IoDeleteSymbolicLink(&Uni);
78099|     //NTFS_EndFiltering(VirtualDrive);
78100|
78101|     // this SHOULD delete the actual snapshot from
78102|     | memory, as no
78103|     // more instances should be using it (ie we are
78104|     | first to use, and last to free)
78105|     if(DevExt->SnapShot) {
78106|         DoneWithSnapShot(DevExt->SnapShot);
78107|         DevExt->SnapShot = NULL;
78108|     }
78109|     DevExt->MasterSnapShot=NULL;
78110| } __except
78111| | (ExceptionFilter(GetExceptionInformation())) {
78112|     Debug(DEBUG_DCPSM,("FreePsmVolume: Exception
78113|     | %08x",GetExceptionCode()));
78114| }
78115|
78116| return STATUS_SUCCESS;
78117| }
78118|
78119| /*
78120| This is called by snapback to dismount a volume.
78121| We really can not delete the devices, as NT does not
78122| | like it, so
78123| we will just mark them as not active, and delete the
78124| | drive letter
78125| associated with them.
78126| */
78127| /*-----*/
78128| | -----*/
78129| NTSTATUS TdDelDrive ( PDEVICE_OBJECT DeviceToNotPSM,

```

```

    | tkSnapShotMaster *MasterSnapShot )
78121| {
78122|     NTSTATUS Status=STATUS_UNSUCCESSFUL;
78123|
78124|     __try {
78125|         PDEVICE_OBJECT DevObj =
            | PSMAN_DRIVER_OBJECT->DeviceObject;
78126|         PFILTERED_EXTENSION
            | FiltExt=(PFILTERED_EXTENSION)
            | GetDeviceExtension(DeviceToNotPSM);
78127|
78128|
            | ASSERT(FiltExt->ObjectType==OBJECT_FILTEREDDISK);
78129|         ASSERT(MasterSnapShot!=NULL);
78130|
78131|         #ifdef DEBUG
78132|             if(!IsSnapShotAcquiredForWrite()) {
78133|                 Debug(DEBUG_DCPSM,("TdDelDrive: Snapshot
            | resource not acquired!\n"));
78134|                 DbgBreakPoint();
78135|             }
78136|         #endif
78137|
78138|         // dont free resources while in use....
78139|         Debug(DEBUG_VDISK,("VDisk: Devcon: Acquiring
            | VDisk resource to free master snapshot %08x for device
            | %08x\n",MasterSnapShot,DeviceToNotPSM));
78140|         AcquireVDiskResource();
78141|
78142|         __try {
78143|             while(DevObj!=NULL) {
78144|                 PDEVICE_OBJECT Next =
            | DevObj->NextDevice;
78145|
78146|
            | if(PsmGetObjectype(DevObj)==OBJECT_VIRTUALDISK) {
78147|                 PVDISK_EXTENSION DevExt =
            | (PVDISK_EXTENSION)GetDeviceExtension(DevObj);
78148|
78149|                 ASSERT(DevObj ==
            | DevExt->DeviceObject);
78150|
78151|                 if ((DevExt->PartitionActive) &&
78152|                     | (DevExt->PSMDevice==DeviceToNotPSM) &&
78153|                     | (MasterSnapShot ==
            | DevExt->MasterSnapShot)) {
78154|                     Status =
            | TdDelVirtualDrive(DevObj);
78155|                     break;

```



```

78156|         }
78157|     }
78158|
78159|         DevObj = Next;
78160|     }
78161| } __finally {
78162|     ReleaseVDiskResource();
78163| }
78164|
78165| } __except
    | (ExceptionFilter(GetExceptionInformation())) {
78166|     Status = GetExceptionCode();
78167|     Debug(DEBUG_DCPSM,("TdDelDrive: Exception
    | %08x",Status));
78168| }
78169| return Status;
78170| }
78171|
78172| ULONG FindEntryForDO( pOpenTransactionInInternal In,
    | PDEVICE_OBJECT DeviceObject)
78173| {
78174|     PFILTERED_EXTENSION
    | DevExt=(PFILTERED_EXTENSION)GetDeviceExtension(DeviceObj
    | ect);
78175|     ULONG i;
78176|
78177|     for(i=0;i<In->NumberOfDevices;i++) {
78178|         if(_wcsicmp((WCHAR
    | *)DN_MakePointer(In,In->DeviceName[i]),DevExt->Name)==0)
    | {
78179|             return i;
78180|         }
78181|     }
78182|     return (ULONG)-1;
78183| }
78184|
78185| /*-----
    | -----*/
78186| NTSTATUS VDiskMapInDrives(tkSnapShotMaster
    | *MasterSnapShot, pOpenTransactionInInternal In, ULONG
    | OTOSize, pOpenTransactionOutInternal Out)
78187| {
78188|     NTSTATUS Status = STATUS_SUCCESS;
78189|     PDEVICE_OBJECT VirtualObject = 0;
78190|     CHAR AssignedInstance = 0;
78191|
78192|     MasterSnapShot->Instance = VDiskInstance;
78193|
78194|     Out->NumberOfDevices = In->NumberOfDevices;
78195|     WCHAR *Buffer =

```

```

    | (WCHAR*)((char*)Out->DeviceName)+Out->NumberOfDevices*si
    | zeof(ULONG);
78196|   OTOSize -= ((char*)Buffer-(char*)Out);
78197|
78198|   // sample of vdisk device name:
78199|   //
    | \Device\PsmDevices_0110\_Device_HarddiskDmVolumes_W2kser
    | ver1Dg0_Volume1_0
78200|
78201|   Debug(DEBUG_VDISK,("VDisk: Mapping in drives for
    | snapshot %08x\n",MasterSnapShot));
78202|   PersistentDictionary::BeginUpdate();
78203|   __try {
78204|       GetSnapShotForWrite();
78205|       __try {
78206|           if(MasterSnapShot) {
78207|               pkSnapShotEntry
    | p=GetTopSnapShotForMaster(&MasterSnapShot->SnapShots);
78208|               while(p) {
78209|
    | if(PsmGetObjectTypes(p->DeviceObject)==OBJECT_FILTEREDDIS
    | K) {
78210|
    | PFILTERED_EXTENSION DevExt =
    | (PFILTERED_EXTENSION)GetDeviceExtension(p->DeviceObject)
    | ;
78211|
    | PRTL_BITMAP CachingBitMap =
    | ((pPersistentDictionary)p->Dictionary)->GetVolumeCaching
    | Map(1);
78212|
78213|
    | // we are depending on
    | SetVolume/SetVolumeInternal having already deactivated
    | the
78214|
    | // Shared->Map by NULLing it
    | after saving it in Shared->MapInTransForm.
78215|
    | ASSERT(
    | !((pPersistentDictionary)p->Dictionary)->GetVolumeCachin
    | gMap(0) );
78216|
78217|
    | ASSERT(IsValidHandle(DevExt->Cache.HeaderFile.FileHandle
    | ));
78218|
    | ASSERT(IsValidHandle(DevExt->Cache.IndexFile.FileHandle)
    | );
78219|
    | ASSERT(IsValidHandle(DevExt->Cache.CacheFile.FileHandle)
    | );
78220|
78221|
    | Debug(DEBUG_VDISK,("VDisk:
    | Mapping in drive %p (%S) for snapshot %08x

```

```

    | %08x\n",p->DeviceObject,DevExt->Name,MasterSnapShot,p));
78222|         Status = TdAddDrive(
    | &CachingBitMap,
    | p->DeviceObject,MasterSnapShot,&VirtualObject,AssignedIn
    | stance);
78223|         AssignedInstance = 1;
78224|         if(NT_SUCCESS(Status)) {
78225|             PVDISK_EXTENSION VdiskExt =
    | (PVDISK_EXTENSION)GetDeviceExtension(VirtualObject);
78226|             ULONG i;
78227|             ULONG Len;
78228|
78229|             // Adding the drive has
    | brought what we know is ( - since we're here!) the
78230|             // live caching bitmap up
    | to date. So we can return it into service.
78231|
78232|             // so - move
    | Shared->MapInTransform back into Shared->Map
78233|
    | ((pPersistentDictionary)p->Dictionary)->SetVolumeCaching
    | Map( 0, CachingBitMap );
78234|             // ... and clear
    | MapInTransform.
78235|
    | ((pPersistentDictionary)p->Dictionary)->SetVolumeCaching
    | Map( 1, NULL);
78236|
78237|             // now that we now the
    | instance, tell the dictionary
78238|             // we it can re map to us
    | during bootup
78239|
    | ((pPersistentDictionary)VdiskExt->SnapShot->Dictionary)-
    | >SetSnapShotInfo(MasterSnapShot);
78240|
78241|
    | Debug(DEBUG_DICT,("Dictionary %08x is snapshot %08x
    | instance %d for %08x\n",
78242|
    | VdiskExt->SnapShot->Dictionary,VdiskExt->DeviceObject,Ma
    | sterSnapShot->Instance,VdiskExt->PSMDevice));
78243|
78244|             i =
    | FindEntryForDO(In,p->DeviceObject);
78245|             if(i!=-1) {
78246|                 Len =
    | (wcslen(VdiskExt->Name)+0x1b)*sizeof(WCHAR)+sizeof(WCHAR
    | );
78247|                 if(Len>OTOSize) {

```

```

78248|                Status =
| STATUS_INVALID_BUFFER_SIZE;
78249|
| Debug(DEBUG_VDISK,("Buffer too small for
| '%S'\n",VdiskExt->Name));
78250|                goto
| ErrorDuringAdd;
78251|                }
78252|                Out->DeviceName[i] =
| DN_MakeOffset(Out,Buffer);
78253|
78254|                //Don says: Changed
| this so that temporary snapshots would get GUID-style
| device names.
78255|                // See commented out
| code immediately below.
78256|
| swprintf(Buffer,L"%s",VdiskExt->VolumeGuid);
78257|
78258|                /*
78259|                //Don says: This seems
| to be messing up temporary snapshots!
78260|                //OTM fossil
78261|                //
| 12345678 9abcdef-12345678 9a
78262|
| if(In->InternalFlags & PSM_IFLAG_PERSISTENT) {
78263|
| swprintf(Buffer,L"%s",VdiskExt->VolumeGuid);
78264|
| } else {
78265|
| swprintf(Buffer,L"\\Device\\PsmDevices_%04x\\%s_%d",PSM_
| LOW_COMPATIBLE_VERSION,VdiskExt->Name,MasterSnapShot->In
| stance);
78266|
| }
78267|                */
78268|
78269|                // get real length.
| which should be less than calculated above.
78270|                Len =
| wcslen(Buffer)*sizeof(WCHAR)+sizeof(WCHAR);
78271|                Buffer = (WCHAR *)
| (((char*)Buffer) + Len);
78272|                OTOSize-=Len;
78273|                } else {
78274|
| Debug(DEBUG_VDISK,("Unable to find entry for
| '%S'\n",DevExt->Name));

```

```

78275|         }
78276|     } else {
78277|         Debug(DEBUG_VDISK,("VDisk:
| failed to map in drive %p, unmapping
| all.\n",p->DeviceObject));
78278|     ErrorDuringAdd:
78279|         ReleaseSnapShotForWrite();
78280|
| VDiskUnMapInDrives(MasterSnapShot);
78281|         GetSnapShotForWrite();
78282|         // free snapshot, as break
| will exit the while loop
78283|         DoneWithSnapShot(p);
78284|         break;
78285|     }
78286| }
78287|
| p=GetNextSnapShotForMaster(&MasterSnapShot->SnapShots,p)
| ;
78288|     }
78289| } else {
78290|     Debug(DEBUG_VDISK,("Map: No Master
| SnapShot work on\n"));
78291| #ifdef DEBUG
78292|         DbgBreakPoint();
78293| #endif
78294|     }
78295| } __finally {
78296|     ReleaseSnapShotForWrite();
78297| }
78298| } __finally {
78299|     PersistentDictionary::EndUpdate();
78300| }
78301|
78302| if(MasterSnapShot->Instance == VDiskInstance) {
78303|     VDiskInstance++;
78304| }
78305|
78306| return Status;
78307| }
78308|
78309| /*-----
| -----*/
78310| NTSTATUS VDiskUnMapInDrives(tkSnapShotMaster
| *MasterSnapShot)
78311| {
78312|     pkSnapShotEntry p;
78313|
78314|     if(!MasterSnapShot)
78315|         return STATUS_INVALID_PARAMETER;

```

```

78316|
78317| //DbgBreakPoint();
78318|
78319| GetSnapShotForWrite();
78320| __try {
78321|     if(MasterSnapShot) {
78322|
78323|         | p=GetTopSnapShotForMaster(&MasterSnapShot->SnapShots);
78324|         while(p) {
78325|             | if(PsmGetObjectTypes(p->DeviceObject)==OBJECT_FILTEREDDISK
78326|             | K) {
78327|                 PFILTERED_EXTENSION DevExt =
78328|                 | (PFILTERED_EXTENSION)GetDeviceExtension(p->DeviceObject)
78329|                 | ;
78330|                 Debug(DEBUG_VDISK,("VDisK:
78331|                 | UnMapping in drive %p (%S) for snapshot %08x
78332|                 | %08x\n",p->DeviceObject,DevExt->Name,MasterSnapShot,p));
78333|                 | TdDelDrive(p->DeviceObject,MasterSnapShot);
78334|                 }
78335|             | p=GetNextSnapShotForMaster(&MasterSnapShot->SnapShots,p)
78336|             | ;
78337|         }
78338|     } else {
78339|         Debug(DEBUG_VDISK,("UnMap: No SnapShot
78340|         | master to work on\n"));
78341|     }
78342| }
78343| #ifdef DEBUG
78344|     DbgBreakPoint();
78345| #endif
78346| } __finally {
78347|     ReleaseSnapShotForWrite();
78348| }
78349| return STATUS_SUCCESS;
78350| }
78351| NTSTATUS VDiskUnMapInAllDrives()
78352| {
78353|     PDEVICE_OBJECT DevObj =
78354|     | PSMAN_DRIVER_OBJECT->DeviceObject;
78355|     PK_SNAPSHOT_ENTRY p;
78356|
78357|     while(DevObj) {
78358|         | if(PsmGetObjectTypes(DevObj)==OBJECT_FILTEREDDISK) {
78359|             PFILTERED_EXTENSION DevExt =
78360|             | (PFILTERED_EXTENSION)GetDeviceExtension(DevObj);
78361|

```

```

78352|         GetSnapShotForWrite();
78353|         __try {
78354|             p=GetTopSnapShot(&DevExt->SnapShots);
78355|             while(p) {
78356|
78357|                 | ASSERT(DevObj==DevExt->DeviceObject);
78358|
78359|                 | if(TdDelDrive(DevObj,p->MasterSnapShot)==STATUS_SUCCESS)
78360|                 | {
78361|                     // start back at top since a
78362|                     | device was deleted
78363|                     DoneWithSnapShot(p);
78364|                     | p=GetTopSnapShot(&DevExt->SnapShots);
78365|                 } else {
78366|                     // This is wierd. We have a
78367|                     | snapshot from a device.
78368|                     // however when we went to look
78369|                     | up the snapshot from the devices
78370|                     // it failed (TdDelDrive only
78371|                     | returns if it deleted it or not, and
78372|                     // the only way it can not have
78373|                     | deleted it is that it didnt find it)
78374|                     // instead of hanging, lets
78375|                     | just skip over it.
78376|                     | Debug(DEBUG_DCPSM,("VDiskUnMapInAllDrives: Invalid
78377|                     | snapshot %08x in %08x\n",p,DevObj));
78378| #ifdef DEBUG
78379|                     DbgBreakPoint();
78380| #endif
78381|                 | p=GetNextSnapShot(&DevExt->SnapShots,p);
78382|             }
78383|             }
78384|             InitializeListHead(&DevExt->SnapShots);
78385|         } __finally {
78386|             ReleaseSnapShotForWrite();
78387|         }
78388|     }
78389|     DevObj = DevObj->NextDevice;
78390| }
78391|
78392| return STATUS_SUCCESS;
78393| }
78394|
78395|
78396|
78397|

```

```

78389| File Listing: VDISK.h
78390|
78391| const ULONG PSM_VDISK_FLAG_READ_MASK
    | = 0x00ff;
78392| const ULONG PSM_VDISK_FLAG_WRITE_MASK
    | = 0xff00;
78393|
78394| const ULONG PSM_VDISK_FLAG_FILL_MASK
    | = 0x0003;
78395| const ULONG PSM_VDISK_FLAG_BUFFER_NO_FILL
    | = 0x0001;
78396| const ULONG PSM_VDISK_FLAG_BUFFER_FILL_WITH_ZEROES
    | = 0x0002;
78397| const ULONG PSM_VDISK_FLAG_BUFFER_FILL_COMPRESS
    | = 0x0003;
78398|
78399| const ULONG PSM_VDISK_FLAG_TACITLY_SUCCEED_WRITES
    | = 0x0100;
78400| const ULONG PSM_VDISK_FLAG_ALLOW_OPEN_FOR_WRITE
    | = 0x0200;
78401| const ULONG PSM_VDISK_FLAG_ALLOW_PSM_FILE_OPEN
    | = 0x0400;
78402| const ULONG PSM_VDISK_FLAG_CACHE_FULL_DELETE_SS
    | = 0x0800;
78403|
78404|
78405| NTSTATUS VDiskMapInDrives(tkSnapshotMaster
    | *MasterSnapShot, pOpenTransactionInInternal In, ULONG
    | OTOSize, pOpenTransactionOutInternal Out);
78406| NTSTATUS VDiskUnMapInDrives(tkSnapshotMaster
    | *MasterSnapShot);
78407| NTSTATUS VDiskDisableAll(void);
78408| NTSTATUS TdDelDrive ( PDEVICE_OBJECT DeviceToNotPSM,
    | tkSnapshotMaster *MasterSnapShot );
78409| NTSTATUS VDiskUnMapInAllDrives(void);
78410| NTSTATUS TdDelVirtualDrive ( PDEVICE_OBJECT
    | VirtualDrive );
78411| NTSTATUS TdAddDrive ( PRTL_BITMAP *CachingBitMap,
    | PDEVICE_OBJECT DeviceToPSM, tkSnapshotMaster
    | *MasterSnapShot, PDEVICE_OBJECT *VirtualDeviceObject,
    | CHAR AssignedInstance );
78412|
78413| extern volatile ULONG VDiskInstance;
78414|
78415|
78416|
78417| File Listing: virgin.cpp
78418|
78419| #include "precomp.h"
78420|

```





```

    | ClusterSizeInBytes);
78459|         ASSERT (GRANULE_SIZE %
    | ClusterSizeInBytes == 0);
78460|         const ULONG
    | ClustersPerGranule = GRANULE_SIZE / ClusterSizeInBytes;
78461|         CLUSTER_INDEX firstCluster
    | = 0;
78462|         CLUSTER_INDEX lastCluster =
    | 0;
78463|         CLUSTER_INDEX numClusters =
    | 0;
78464|
78465|         GetSnapshotForRead();
78466|         __try {
78467|             pDictionary
    | AnyDictionaryOnVolume = 0;
78468|             GetDictionaryForVolume
    | (Volume, AnyDictionaryOnVolume);
78469|             if (
    | AnyDictionaryOnVolume ) {
78470|                 pShared Shared =
    | ((pPersistentDictionary)AnyDictionaryOnVolume)->Shared;
78471|                 status =
    | FindVirginSpace_GranuleBitmapPhase (VirginMap, Shared,
    | ClustersPerGranule, NumExtentsLost, TotalClusters);
78472|                 if (
    | NT_SUCCESS(status) ) {
78473|                     status =
    | FindVirginSpace_SnapshotPhase (VirginMap, TempMap,
    | Shared, ClustersPerGranule, NumExtentsLost);
78474|                     if (
    | NT_SUCCESS(status) ) {
78475|                         status =
    | FindVirginSpace_VolumeBitmapPhase(VirginMap, TempMap,
    | NumExtentsLost, VolumeObject);
78476|                     }
78477|                 }
78478|             } else {
78479|
    | VirginDebug(("pd::FindVirginSpace: No persistent
    | dictionary was found on volume %08x\n",Volume));
78480|
    | // caller can use
    | this return code to know that cache file doesn't need
    | to be migrated
78482|             status =
    | PSM_VIRGIN_ERROR_NO_SNAPSHOTS;
78483|         }
78484|     } __finally {
78485|

```

```

    | ReleaseSnapShotForRead();
78486|         }
78487|     }
78488|     } __finally {
78489|         Sblo_CloseVolumeHandle
    | (VolumeHandle, VolumeObject);
78490|     }
78491|     }
78492|     } __finally {
78493|         if ( AbnormalTermination() ) {
78494|             TempMap.reset(); // if exception
    | did not occur, destructor will clean up TempMap.
78495|         }
78496|     }
78497|     } else {
78498|         status = STATUS_INVALID_PARAMETER;
78499|         VirginDebug(("pd::FindVirginSpace: invalid
    | parameter: device object is not a filtered disk!\n"));
78500|         ASSERT(FALSE);
78501|     }
78502| } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
78503|     status = GetExceptionCode();
78504|     VirginDebug(("!!! pd::FindVirginSpace:
    | exception %08x\n",status));
78505| }
78506|
78507| VirginDebug(("pd::FindVirginSpace:
    | NumExtentsLost=%08x, returning
    | status=%08x\n",NumExtentsLost,status));
78508| return status;
78509| }
78510|
78511| //-----
    | -----
78512|
78513| NTSTATUS
    | PersistentDictionary::FindVirginSpace_GranuleBitmapPhase
    | (
78514|     tVirginMap    &VirginMap,
78515|     pShared       Shared,
78516|     ULONG         ClustersPerGranule,
78517|     ULONG         &NumExtentsLost,
78518|     LARGE_INTEGER TotalClusters )
78519| {
78520|     // For every run of zeroes in the granule bitmap,
    | insert the clusters in the
78521|     // run into VirginMap.
78522|
78523|     Profile("pd::FindVirginSpace_GranulueBitmapPhase");

```

```

78524|
| VirginDebug(("pd::FindVirginSpace_GranuleBitmapPhase
| called: Shared=%08x, ClustersPerGranule=%08x,
| TotalClusters=%016l64x\n",Shared,ClustersPerGranule,Tota
| IClusters.QuadPart));
78525|   if ( Shared ) {
78526|
| VirginDebug(("pd::FindVirginSpace_GranuleBitmapPhase:
| Shared->Map = %08x\n",Shared->Map));
78527|   }
78528|
78529|   NTSTATUS status = STATUS_SUCCESS;
78530|
78531|   if ( DoFreeSpaceChecks() ) {
78532|       if ( Shared && Shared->Map ) {
78533|           ULONG HintIndex = 0;
78534|           ULONG BitIndexStart = 0;
78535|           while ( BitIndexStart!=INVALID_BIT_INDEX &&
| NT_SUCCESS(status) &&
| HintIndex<(Shared->Map->SizeOfBitMap) ) {
78536|               PsmBitMapValidate (Shared->Map);
78537|               BitIndexStart = RtlFindClearBits
| (Shared->Map, 1, HintIndex);
78538|               VirginDebug((" pd::fvs_gbp:
| BitIndexStart = %08x\n",BitIndexStart));
78539|               if ( BitIndexStart != INVALID_BIT_INDEX
| ) {
78540|                   if ( BitIndexStart >= HintIndex ) {
78541|                       ULONG BitIndexEnd =
| RtlFindSetBits (Shared->Map, 1, BitIndexStart);
78542|                       VirginDebug((" pd::fvs_gbp:
| (A) BitIndexEnd = %08x\n",BitIndexEnd));
78543|                       if ( BitIndexEnd ==
| INVALID_BIT_INDEX ) {
78544|                           BitIndexEnd =
| Shared->Map->SizeOfBitMap;
78545|                           VirginDebug(("
| pd::fvs_gbp: (B) INVALID_BIT_INDEX ->
| BitIndexEnd=%08x\n",BitIndexEnd));
78546|                       } else if ( BitIndexEnd <
| BitIndexStart ) {
78547|                           BitIndexEnd =
| Shared->Map->SizeOfBitMap;
78548|                           VirginDebug(("
| pd::fvs_gbp: (C) RtlFindSetBits wrapped around ->
| BitIndexEnd=%08x\n",BitIndexEnd));
78549|                       }
78550|
78551|                       HintIndex = 1+BitIndexEnd;
78552|                       ULONG NumGranulesClear =

```

```

    | (BitIndexEnd - BitIndexStart);
78553|         CLUSTER_INDEX firstCluster =
    | BitIndexStart * ClustersPerGranule;
78554|         CLUSTER_INDEX numClusters =
    | NumGranulesClear * ClustersPerGranule;
78555|
78556|         // Now need to make sure we
    | aren't going past end of the volume, based on
    | TotalClusters.
78557|         // Assuming we get the correct
    | range of extent values in this function, all the others
78558|         // should stay in that range
    | because they just fragment and/or re-insert extents.
78559|
78560|         if ( firstCluster <
    | TotalClusters.QuadPart ) {
78561|             CLUSTER_INDEX lastCluster =
    | firstCluster + numClusters - 1;
78562|             ASSERT(lastCluster >=
    | firstCluster);
78563|             if ( lastCluster >=
    | TotalClusters.QuadPart ) {
78564|                 lastCluster =
    | TotalClusters.QuadPart - 1;
78565|                 ASSERT(lastCluster >=
    | firstCluster);
78566|                 VirginDebug(("
    | pd::fvs_gbp: Truncating numClusters=%016l64x to
    | %016l64x\n",numClusters,lastCluster-firstCluster+1));
78567|                 numClusters =
    | lastCluster - firstCluster + 1;
78568|             }
78569|             status =
    | VirginMap.insertExtent (firstCluster, numClusters);
78570|             if ( status ==
    | PSM_TREE_INSERT_ERROR ) {
78571|                 ++NumExtentsLost;
78572|                 status =
    | STATUS_SUCCESS; // not severe enough error to give
    | up
78573|             }
78574|             } else {
78575|                 VirginDebug(("
    | pd::fvs_gbp: firstCluster=%016l64x is past end of
    | volume - ignoring\n",firstCluster));
78576|             }
78577|             } else {
78578|                 VirginDebug((" pd::fvs_gbp:
    | RtlFindClearBits wrapped around.\n"));
78579|                 break;

```

```

78580|         }
78581|     }
78582| }
78583|     } else {
78584|         status =
            | PSM_VIRGIN_ERROR_FREE_SPACE_DISABLED;
78585|         | VirginDebug(("pd::FindVirginSpace_GranuleBitmapPhase:
            | granule bitmap is not available!!!\n"));
78586|         ASSERT(FALSE);
78587|     }
78588| } else {
78589|     | VirginDebug(("pd::FindVirginSpace_GranuleBitmapPhase:
            | free space checking is disabled!\n"));
78590|     status = PSM_VIRGIN_ERROR_FREE_SPACE_DISABLED;
78591| }
78592|
78593|     | VirginDebug(("pd::FindVirginSpace_GranuleBitmapPhase:
            | Dump of VirginMap:\n"));
78594|     VirginMap.debugDump();
78595|
78596|     | VirginDebug(("pd::FindVirginSpace_GranuleBitmapPhase
            | returning %08x\n",status));
78597|     return status;
78598| }
78599|
78600| //-----
            | -----
78601|
78602| NTSTATUS
            | PersistentDictionary::FindVirginSpace_SnapshotPhase (
78603|     tVirginMap &VirginMap,
78604|     tVirginMap &TempMap,
78605|     pShared Shared,
78606|     ULONG ClustersPerGranule,
78607|     ULONG &NumExtentsLost )
78608| {
78609|     // !!! Now remove extents from VirginMap, fragment
            | as necessary based on
78610|     // snapshot usage and virtual writes in snapshots,
            | then insert
78611|     // the results one by one into TempMap.
78612|
78613|     Profile("pd::FindVirginSpace_SnapshotPhase");
78614|     VirginDebug(("pd::FindVirginSpace_SnapshotPhase
            | called\n"));
78615|

```

```

78616| NTSTATUS status = STATUS_SUCCESS;
78617| CLUSTER_INDEX firstCluster = 0;
78618| CLUSTER_INDEX numClusters = 0;
78619| CLUSTER_INDEX lastCluster = 0;
78620|
78621| while ( NT_SUCCESS(status) && !VirginMap.isEmpty()
| ) {
78622|     status = VirginMap.removeAnyExtent (
| firstCluster, numClusters );
78623|     ASSERT(NT_SUCCESS(status));
78624|     if ( NT_SUCCESS(status) ) {
78625|         CLUSTER_INDEX originalLastCluster =
| firstCluster + numClusters - 1;
78626|         CLUSTER_INDEX firstGranule = firstCluster /
| ClustersPerGranule;
78627|         CLUSTER_INDEX lastGranule =
| originalLastCluster / ClustersPerGranule;
78628|         ASSERT(firstGranule <= 0xffffffff);
78629|         ASSERT(lastGranule <= 0xffffffff);
78630|         ASSERT(lastGranule >= firstGranule);
78631|         LARGE_INTEGER Key = {0};
78632|         CLUSTER_INDEX granule = firstGranule;
78633|         while ( granule <= lastGranule &&
| NT_SUCCESS(status) ) {
78634|             bool provedClear = true;
78635|             Key.GranulePart = ULONG(granule);
78636|             Key.SnapShotPart = 0;
78637|             MyAcquireResourceSharedLite
| (&(Shared->TreeResource), TRUE);
78638|             __try {
78639|                 tTreeLeaf *node1 =
| rbtree_SearchUpperBound (&Shared->Tree, Key.QuadPart);
78640|                 tTreeLeaf *node2 =
| rbtree_SearchUpperBound (&Shared->VirtualWritesTree,
| Key.QuadPart);
78641|                 tTreeLeaf *relevantNode = node1;
78642|                 if ( node1 ) {
78643|                     if ( node2 ) {
78644|                         LARGE_INTEGER Key1, Key2;
78645|                         Key1.QuadPart = node1->Key;
78646|                         Key2.QuadPart = node2->Key;
78647|                         if ( Key2.GranulePart <
| Key1.GranulePart ) {
78648|                             relevantNode = node2;
78649|                         }
78650|                     }
78651|                 } else {
78652|                     relevantNode = node2;
78653|                 }
78654|

```

```

78655|             if ( relevantNode ) {
78656|                 LARGE_INTEGER rKey;
78657|                 rKey.QuadPart =
| relevantNode->Key;
78658|                 CLUSTER_INDEX granuleHit =
| CLUSTER_INDEX (rKey.GranulePart);
78659|                 if ( granuleHit <= lastGranule
| ) {
78660|                     if ( granuleHit > granule )
| {
78661|                         // There is a fragment
| [granule..(granuleHit-1)]
78662|                         firstCluster = granule
| * ClustersPerGranule;
78663|                         lastCluster =
| (granuleHit-1)*ClustersPerGranule +
| (ClustersPerGranule-1);
78664|                         if ( lastCluster >
| originalLastCluster ) {
78665|                             lastCluster =
| originalLastCluster;
78666|                         }
78667|                         ASSERT(lastCluster >=
| firstCluster);
78668|                         numClusters =
| lastCluster - firstCluster + 1;
78669|                         status =
| TempMap.insertExtent (firstCluster, numClusters);
78670|                         if ( status ==
| PSM_TREE_INSERT_ERROR ) {
78671|                             ++NumExtentsLost;
78672|                             status =
| STATUS_SUCCESS; // not severe enough error to give
| up
78673|                         }
78674|                     }
78675|
78676|                     granule = 1 + granuleHit;
| // continue search after the interruption
78677|                     provedClear = false;
78678|                 }
78679|             }
78680|         } __finally {
78681|             MyReleaseResourceForThreadLite
| (&(Shared->TreeResource));
78682|         }
78683|
78684|         if ( provedClear ) {
78685|             // No more interruptions exist in
| this granule extent!

```



```

78686|         firstCluster = granule *
| ClustersPerGranule;
78687|         lastCluster =
| lastGranule*ClustersPerGranule +
| (ClustersPerGranule-1);
78688|         if ( lastCluster >
| originalLastCluster ) {
78689|             lastCluster =
| originalLastCluster;
78690|         }
78691|         ASSERT(lastCluster >=
| firstCluster);
78692|         numClusters = lastCluster -
| firstCluster + 1;
78693|         status = TempMap.insertExtent
| (firstCluster, numClusters);
78694|         if ( status ==
| PSM_TREE_INSERT_ERROR ) {
78695|             ++NumExtentsLost;
78696|             status = STATUS_SUCCESS; //
| not severe enough error to give up
78697|         }
78698|         break; // move on to the next
| granule extent
78699|     }
78700| }
78701| }
78702| }
78703|
78704| VirginDebug(("pd::FindVirginSpace_SnapshotPhase:
| Dump of TempMap:\n"));
78705| TempMap.debugDump();
78706|
78707| VirginDebug(("pd::FindVirginSpace_SnapshotPhase:
| Dump of VirginMap:\n"));
78708| VirginMap.debugDump();
78709|
78710| VirginDebug(("pd::FindVirginSpace_SnapshotPhase:
| NumExtentsLost=%08x, returning
| status=%08x\n",NumExtentsLost,status));
78711| return status;
78712| }
78713|
78714| //-----
| -----
78715|
78716| NTSTATUS
| PersistentDictionary::FindVirginSpace_VolumeBitmapPhase
| (
78717|     tVirginMap    &VirginMap,

```

```

78718| tVirginMap    &TempMap,
78719| ULONG         &NumExtentsLost,
78720| PFILE_OBJECT  VolumeObject )
78721| {
78722| Profile("pd::FindVirginSpace_VolumeBitmapPhase");
78723| VirginDebug(("pd::FindVirginSpace_VolumeBitmapPhase
| called\n"));
78724|
78725| NTSTATUS status = STATUS_SUCCESS;
78726| CLUSTER_INDEX firstCluster = 0;
78727| CLUSTER_INDEX lastCluster = 0;
78728| CLUSTER_INDEX numClusters = 0;
78729|
78730| if ( !TempMap.isEmpty() ) {
78731|     // Allocate space for the volume bitmap window
| based on the largest
78732|     // extent in TempMap. This will be the largest
| bitmap window needed for
78733|     // the rest of this process.
78734|
78735|     status = TempMap.queryLongestExtent
| (numClusters);
78736|     if ( NT_SUCCESS(status) ) {
78737|         // We need one bit in the volume bitmap for
| every cluster.
78738|         // Round up to the nearest number of words.
78739|         ULONG MaxClusters = (ULONG) numClusters;
78740|         ASSERT (CLUSTER_INDEX(MaxClusters) ==
| numClusters);
78741|         ULONG VolumeBitmapWords = (MaxClusters +
| 31) / 32;
78742|         ULONG VolumeBitmapBytes = sizeof(DWORD) *
| VolumeBitmapWords;
78743|         ULONG VolumeBitmapAlloc =
| FIELD_OFFSET(VOLUME_BITMAP_BUFFER,Buffer) +
| VolumeBitmapBytes + sizeof(DWORD);
78744|         VOLUME_BITMAP_BUFFER *WorkUnit =
| (VOLUME_BITMAP_BUFFER *) MemAllocatePoolWithTag(
78745|             PagedPool,
78746|             VolumeBitmapAlloc,
78747|             TEMPTAG );
78748|
78749|         if ( WorkUnit ) {
78750|             __try {
78751|                 // !!! Remove nodes from TempMap,
| fragment as necessary based on live volume bitmap,
78752|                 // and insert the results into
| VirginMap.
78753|
78754|                 RtlZeroMemory (WorkUnit,

```

```

    | VolumeBitmapAlloc);
78755|         RTL_BITMAP WorkUnitBitMap;
78756|         RtlInitializeBitMap (
    | &WorkUnitBitMap, (PULONG)(&WorkUnit->Buffer),
    | MaxClusters );
78757|
78758|         while ( NT_SUCCESS(status) &&
    | !TempMap.isEmpty() ) {
78759|             status =
    | TempMap.removeAnyExtent (firstCluster, numClusters);
78760|             ASSERT(NT_SUCCESS(status));
78761|             ASSERT(numClusters <=
    | MaxClusters);
78762|             ASSERT(numClusters > 0);
78763|             if ( NT_SUCCESS(status) ) {
78764|                 CLUSTER_INDEX virginCluster
    | = firstCluster;
78765|                 lastCluster = firstCluster
    | + numClusters - 1;
78766|                 VirginDebug(("pd::fvs_vbp:
    | --- firstCluster=%016l64x, numClusters=%016l64x,
    | lastCluster=%016l64x\n",firstCluster,numClusters,lastClu
    | ster));
78767|                 ULONG NumBytes =
    | 4*((ULONG(numClusters) + 31) / 32) +
    | FIELD_OFFSET(VOLUME_BITMAP_BUFFER,Buffer);
78768|                 ASSERT(NumBytes <=
    | VolumeBitmapAlloc);
78769|                 STARTING_LCN_INPUT_BUFFER
    | slib = {0};
78770|                 slib.StartingLcn.QuadPart =
    | firstCluster;
78771|                 status = FS_GetVolumeBitmap
    | (VolumeObject, &slib, WorkUnit, NumBytes);
78772|                 if ( status ==
    | STATUS_BUFFER_OVERFLOW ) {
78773|                     status =
    | STATUS_SUCCESS; // not a problem... just means more
    | volume data
78774|                 }
78775|
78776|                 if ( NT_SUCCESS(status) ) {
78777|                     ULONG HintIndex = 0;
78778|                     ULONG LastIndexInWindow
    | = ULONG(numClusters-1);
78779|                     ASSERT (
    | CLUSTER_INDEX(LastIndexInWindow)+1 == numClusters );
78780|                     while (
    | NT_SUCCESS(status) ) {
78781|

```

```

    | VirginDebug(("#loop# virginCluster=%016l64x,
    | HintIndex=%08x\n",virginCluster,HintIndex));
78782|                // See if any
    | clusters in this extent are dirty.
78783|                ULONG DirtyIndex =
    | RtlFindSetBits (&WorkUnitBitMap, 1, HintIndex);
78784|                if (
    | DirtyIndex==INVALID_BIT_INDEX || DirtyIndex<HintIndex
    | || DirtyIndex>LastIndexInWindow ) {
78785|
    | VirginDebug(("#break (A)# DirtyIndex=%08x,
    | HintIndex=%08x,
    | LastIndexInWindow=%08x\n",DirtyIndex,HintIndex,LastIndex
    | InWindow));
78786|                break; // no
    | more dirty clusters in the volume bitmap window
78787|                } else {
78788|                CLUSTER_INDEX
    | DirtyCluster = firstCluster + DirtyIndex;
78789|                if (
    | DirtyCluster <= lastCluster ) {
78790|                if (
    | DirtyCluster > virginCluster ) {
78791|
    | CLUSTER_INDEX numCleanClusters = DirtyCluster -
    | virginCluster;
78792|
    | VirginDebug(("pd::fvs_vbp: +++
    | virginCluster=%016l64x, DirtyCluster=%016l64x,
    | numCleanClusters=%016l64x\n",virginCluster,DirtyCluster,
    | numCleanClusters));
78793|                status
    | = VirginMap.insertExtent (virginCluster,
    | numCleanClusters);
78794|                if (
    | status == PSM_TREE_INSERT_ERROR ) {
78795|
    | ++NumExtentsLost;
78796|
    | status = STATUS_SUCCESS;
78797|                }
78798|                } else if (
    | DirtyCluster != virginCluster ) {
78799|
    | ASSERT(FALSE);    // should not happen, but prevent
    | infinite loop potential
78800|                status
    | = STATUS_UNSUCCESSFUL;
78801|                break;
78802|                }

```

```

78803|
78804|             HintIndex =
| RtlFindClearBits (&WorkUnitBitMap, 1, DirtyIndex);
78805|             if (
| HintIndex==INVALID_BIT_INDEX || HintIndex<DirtyIndex ||
| HintIndex>LastIndexInWindow ) {
78806|
| VirginDebug(("#break (B)# DirtyIndex=%08x,
| HintIndex=%08x,
| LastIndexInWindow=%08x\n",DirtyIndex,HintIndex,LastIndex
| InWindow));
78807|
| virginCluster = 1+lastCluster; // prevent inserting
| tail extent
78808|             break;
| // no more clean clusters in this extent
78809|             } else {
78810|
| virginCluster = firstCluster + HintIndex;
78811|             }
78812|             } else {
78813|
| VirginDebug(("#break (C)# DirtyCluster=%016l64x,
| lastCluster=%016l64x\n",DirtyCluster,lastCluster));
78814|             break; //
| no more dirty clusters in this extent
78815|             }
78816|             }
78817|             }
78818|
78819|             if ( NT_SUCCESS(status)
| ) {
78820|                 if ( lastCluster >=
| virginCluster ) {
78821|                     numClusters =
| lastCluster - virginCluster + 1;
78822|
| VirginDebug(("pd::fvs_vbp: +++
| virginCluster=%016l64x, numClusters=%016l64x,
| lastCluster=%016l64x\n",virginCluster,numClusters,lastCl
| uster));
78823|                     status =
| VirginMap.insertExtent (virginCluster, numClusters);
78824|                     if ( status ==
| PSM_TREE_INSERT_ERROR ) {
78825|
| ++NumExtentsLost;
78826|                     status =
| STATUS_SUCCESS;
78827|             }

```

```

78828|         }
78829|     }
78830| } else {
78831|     | VirginDebug(("pd::FindVirginSpace_VolumeBitmapPhase:
    | FS_GetVolumeBitmap returned %08x\n",status));
78832|         ASSERT(FALSE);
78833|     }
78834| }
78835| }
78836| } __finally {
78837|     MemFreePool (WorkUnit);
78838|     WorkUnit = 0;
78839| }
78840| } else {
78841|     status = STATUS_INSUFFICIENT_RESOURCES;
78842|     VirginDebug(("!!! pd::FindVirginSpace:
    | Out of memory (volume bitmap)\n"));
78843| }
78844| } else {
78845|     VirginDebug(("!!! pd::FindVirginSpace:
    | TempMap.queryLongestExtent() returned %08x\n",status));
78846|     ASSERT(FALSE);
78847| }
78848| }
78849|
78850|     | VirginDebug(("pd::FindVirginSpace_VolumeBitmapPhase:
    | Dump of VirginMap:\n"));
78851|     VirginMap.debugDump();
78852|
78853|     | VirginDebug(("pd::FindVirginSpace_VolumeBitmapPhase:
    | Dump of TempMap:\n"));
78854|     TempMap.debugDump();
78855|
78856|     | VirginDebug(("pd::FindVirginSpace_VolumeBitmapPhase:
    | NumExtentsLost=%08x, returning
    | status=%08x\n",NumExtentsLost,status));
78857|     return status;
78858| }
78859|
78860| //-----
    | -----
78861|
78862| NTSTATUS GetClusterSize ( PFILTERED_EXTENSION DevExt,
    | ULONG &ClusterSizeInBytes, LARGE_INTEGER &TotalClusters
    | )
78863| {

```

```

78864| Profile("GetClusterSize");
78865| VirginDebug(("GetClusterSize:
| DevExt=%08x\n",DevExt));
78866| NTSTATUS status = STATUS_NOT_FOUND;
78867| ClusterSizeInBytes = 0;
78868| TotalClusters.QuadPart = 0;
78869|
78870| __try {
78871|     pPsmFileInfo table[] = {
78872|         &(DevExt->Cache.CacheFile),
78873|         &(DevExt->Cache.IndexFile),
78874|         &(DevExt->Cache.HeaderFile),
78875|         NULL
78876|     };
78877|
78878|     for ( int i=0; table[i] != NULL; ++i ) {
78879|         if ( IsValidHandle(table[i]->FileObject) )
78880|         | {
78881|             VirginDebug(("GetClusterSize: Calling
| FS_GetVolumeInfo on table[%d]->FileObject=%08x
| '%S'\n",i,table[i]->FileObject,table[i]->FileName));
78882|             ULONG SectorSize = 0;
78883|             LARGE_INTEGER AvailClusters={0};
78884|             NTSTATUS infoStatus = FS_GetVolumeInfo
| (table[i]->FileObject, ClusterSizeInBytes, SectorSize,
| TotalClusters, AvailClusters);
78885|             if ( NT_SUCCESS(infoStatus) ) {
78886|                 VirginDebug(("GetClusterSize:
| Found cluster size %08x from
| FS_GetVolumeInfo\n",ClusterSizeInBytes));
78887|                 status = STATUS_SUCCESS;
78888|                 break;
78889|             } else {
78890|                 VirginDebug(("GetClusterSize:
| FS_GetVolumeInfo returned %08x\n",infoStatus));
78891|             }
78892|         }
78893|     } __except(
| ExceptionFilter(GetExceptionInformation()) ) {
78894|         status = GetExceptionCode();
78895|         VirginDebug(("!!! GetClusterSize: Exception
| %08x\n",status));
78896|     }
78897|
78898|     VirginDebug(("GetClusterSize:
| ClusterSizeInBytes=%08x, returning
| status=%08x\n",ClusterSizeInBytes,status));
78899|     return status;
78900| }

```

```

78901|
78902| //-----
78903|
78904| #ifdef DEBUG
78905| NTSTATUS TestVirginMap_OverwriteExtent (
78906|     PFILTERED_EXTENSION    DevExt,
78907|     const char              *WriteBuffer,
78908|     CLUSTER_INDEX           WriteBufferClusters,
78909|     ULONG                    ClusterSizeInBytes,
78910|     CLUSTER_INDEX           FirstCluster,
78911|     CLUSTER_INDEX           NumClusters )
78912| {
78913|     NTSTATUS status = STATUS_SUCCESS;
78914|     Profile("TestVirginMap_OverwriteExtent");
78915|     VirginDebug((" tvm_OverwriteExtent: DevExt=%08x,
78916|         | FirstCluster=%016l64x,
78917|         | NumClusters=%016l64x\n", DevExt, FirstCluster, NumClusters)
78918|         | );
78919|     CLUSTER_INDEX CurrentCluster = FirstCluster;
78920|     CLUSTER_INDEX ClustersRemaining = NumClusters;
78921|     LARGE_INTEGER ByteOffset = {0};
78922|     while ( ClustersRemaining > 0 ) {
78923|         Profile("TestVirginMap_OverwriteExtent -
78924|             | ClustersRemaining loop");
78925|         CLUSTER_INDEX ClustersToWrite =
78926|             | ClustersRemaining;
78927|         if ( WriteBufferClusters < ClustersToWrite ) {
78928|             ClustersToWrite = WriteBufferClusters;
78929|         }
78930|         ByteOffset.QuadPart = CurrentCluster *
78931|             | ClusterSizeInBytes;
78932|         CLUSTER_INDEX ByteCountLarge = ClustersToWrite
78933|             | * ClusterSizeInBytes;
78934|         ASSERT(ByteCountLarge<=0xffffffff);
78935|         ULONG ByteCount = ULONG(ByteCountLarge);
78936|         VirginDebug((" Overwriting
78937|             | ByteOffset=%016l64x,
78938|             | ByteCount=%08x\n", ByteOffset.QuadPart, ByteCount));
78939|         status = Sblo_WriteDevice
78940|             | (DevExt->DeviceObject, &ByteOffset, ByteCount,
78941|             | WriteBuffer);
78942|         if ( !NT_SUCCESS(status) ) {
78943|             VirginDebug((" Sblo_WriteDevice
78944|                 | returned status=%08x\n", status));
78945|             ASSERT(FALSE);

```



```

78938|         break;
78939|     }
78940|
78941|     CurrentCluster += ClustersToWrite;
78942|     ClustersRemaining -= ClustersToWrite;
78943| }
78944|
78945| VirginDebug((" tvm_OverwriteExtent returning
| status=%08x\n",status));
78946| return status;
78947| }
78948| #endif /*DEBUG*/
78949|
78950| //-----
| -----
78951|
78952| #ifdef DEBUG
78953| void InitVirginMapTestBuffer ( char *WriteBuffer, ULONG
| WriteBufferBytes )
78954| {
78955|     const char *FillPattern = "(Virgin)";
78956|     const ULONG PatternBytes = strlen(FillPattern);
78957|     ASSERT (WriteBufferBytes % PatternBytes == 0);
78958|     ULONG BytesFilled = 0;
78959|     while ( BytesFilled < WriteBufferBytes ) {
78960|         memcpy (&WriteBuffer[BytesFilled], FillPattern,
| PatternBytes);
78961|         BytesFilled += PatternBytes;
78962|     }
78963| }
78964| #endif /*DEBUG*/
78965|
78966| //-----
| -----
78967|
78968| #ifdef DEBUG
78969| NTSTATUS TestVirginMap_OverwriteAllExtents (
78970|     PDEVICE_OBJECT    Volume,
78971|     ULONG              ClusterSizeInBytes,
78972|     LARGE_INTEGER      TotalClusters,
78973|     tVirginMap         &VirginMap )
78974| {
78975|     NTSTATUS status = STATUS_SUCCESS;
78976|     VirginDebug(("Entering
| TestVirginMap_OverwriteAllExtents:
| Volume=%08x\n",Volume));
78977|
78978|     __try {
78979|         if ( PsmGetObjectType(Volume) ==
| OBJECT_FILTEREDDISK ) {

```

```

78980|         PFILTERED_EXTENSION DevExt =
| (PFILTERED_EXTENSION) GetDeviceExtension(Volume);
78981|         tVirginMap TempMap;    // holding pen for
| extents.
78982|         __try {
78983|             CLUSTER_INDEX maxClusters = 0;
78984|             status =
| VirginMap.queryLongestExtent(maxClusters);
78985|             if ( NT_SUCCESS(status) ) {
78986|                 const ULONG WriteBufferClusters =
| 256;
78987|                 const ULONG WriteBufferBytes =
| WriteBufferClusters * ClusterSizeInBytes;
78988|                 VirginDebug(("WriteBufferBytes =
| %08x\n",WriteBufferBytes));
78989|                 char *WriteBuffer = (char
| *)MemAllocatePoolWithTag(PagedPool,WriteBufferBytes,TEMP
| TAG);
78990|                 if ( WriteBuffer ) {
78991|                     __try {
78992|                         CLUSTER_INDEX
| firstCluster=0;
78993|                         CLUSTER_INDEX
| numClusters=0;
78994|                         CLUSTER_INDEX
| prevNumClusters=0;
78995|
78996|                         InitVirginMapTestBuffer
| (WriteBuffer, WriteBufferBytes);
78997|
78998|                         while ( NT_SUCCESS(status)
| && !VirginMap.isEmpty() ) {
78999|                             status =
| VirginMap.removeLongestExtent (firstCluster,
| numClusters);
79000|                             if ( NT_SUCCESS(status)
| ) {
79001|                                 | ASSERT(numClusters>0);
79002|                                 if (
| prevNumClusters ) {
79003|                                     if (
| numClusters > prevNumClusters ) {
79004|                                         | VirginDebug(("!!! (A) numClusters=%016l64x,
| prevNumClusters=%016l64x,
| firstCluster=%016l64x\n",numClusters,prevNumClusters,fir
| stCluster));
79005|                                         | ASSERT(numClusters <= prevNumClusters);

```

```

79006|                status =
| STATUS_UNSUCCESSFUL;
79007|                }
79008|                }
79009|                prevNumClusters =
| numClusters;
79010|                status =
| TempMap.insertExtent (firstCluster, numClusters); //
| save for later
79011|                if (
| NT_SUCCESS(status) ) {
79012|                ASSERT
| (firstCluster < TotalClusters.QuadPart);
79013|                ASSERT
| (firstCluster+numClusters-1 < TotalClusters.QuadPart);
79014|                status =
| TestVirginMap_OverwriteExtent (DevExt, WriteBuffer,
| WriteBufferClusters, ClusterSizeInBytes, firstCluster,
| numClusters);
79015|                } else {
79016|                ASSERT(FALSE);
79017|                status =
| STATUS_UNSUCCESSFUL;
79018|                }
79019|                } else {
79020|                ASSERT(FALSE);
79021|                status =
| STATUS_UNSUCCESSFUL;
79022|                }
79023|                }
79024|
79025|                // put all the stuff from
| TempMap back into VirginMap
79026|                prevNumClusters = 0;
79027|                while ( !TempMap.isEmpty()
| ) {
79028|                NTSTATUS tempStatus =
| TempMap.removeLongestExtent (firstCluster,
| numClusters);
79029|                if (
| NT_SUCCESS(tempStatus) ) {
79030|                tempStatus =
| VirginMap.insertExtent (firstCluster, numClusters);
79031|                | ASSERT(NT_SUCCESS(tempStatus));
79032|                if (
| NT_SUCCESS(tempStatus) ) {
79033|                | ASSERT(numClusters>0);
79034|                if (

```

```

    | prevNumClusters ) {
79035|         if (
    | numClusters > prevNumClusters ) {
79036|
    | VirginDebug(("!!! (B) numClusters=%016l64x,
    | prevNumClusters=%016l64x,
    | firstCluster=%016l64x\n",numClusters,prevNumClusters,fir
    | stCluster));
79037|
    | ASSERT(numClusters<=prevNumClusters);
79038|         }
79039|     }
79040|     prevNumClusters
    | = numClusters;
79041|     }
79042|     } else {
79043|         ASSERT(FALSE);
79044|     }
79045|     }
79046|     } __finally {
79047|         MemFreePool(WriteBuffer);
79048|         WriteBuffer = NULL;
79049|     }
79050|     } else {
79051|         VirginDebug(("!!!
    | TestVirginMap_OverwriteAllExtents: Out of memory
    | !!!\n"));
79052|         status =
    | STATUS_INSUFFICIENT_RESOURCES;
79053|     }
79054|     } else {
79055|         VirginDebug(("!!!
    | TestVirginMap_OverwriteAllExtents:
    | VirginMap.queryLongestExtent() returned
    | %08x\n",status));
79056|         ASSERT(FALSE);
79057|     }
79058|     } __finally {
79059|         TempMap.reset();
79060|     }
79061|     } else {
79062|
    | VirginDebug(("TestVirginMap_OverwriteAllExtents:
    | Volume is not a filtered disk!\n"));
79063|         status = STATUS_INVALID_PARAMETER;
79064|         ASSERT(FALSE);
79065|     }
79066|     } __except(
    | ExceptionFilter(GetExceptionInformation()) ) {
79067|         status = GetExceptionCode();

```

```

79068|     VirginDebug(("!!!
| TestVirginMap_OverwriteAllExtents: Exception
| %08x\n",status));
79069|     }
79070|
79071|     VirginDebug(("TestVirginMap_OverwriteAllExtents:
| returning status=%08x\n",status));
79072|     return status;
79073| }
79074| #endif /*DEBUG*/
79075|
79076| //-----
| -----
79077|
79078| #ifdef DEBUG
79079| void TestVirginMap ( void *parm )
79080| {
79081|     PDEVICE_OBJECT Volume = (PDEVICE_OBJECT)parm;
79082|     VirginDebug(("Entering TestVirginMap() - Volume =
| %08x\n",parm));
79083|
79084|     tVirginMap VirginMap;
79085|     ULONG ClusterSizeInBytes = 0;
79086|     LARGE_INTEGER TotalClusters = {0};
79087|     NTSTATUS status =
| PersistentDictionary::FindVirginSpace (Volume,
| VirginMap, ClusterSizeInBytes, TotalClusters);
79088|     VirginDebug(("TestVirginMap: pd::FindVirginSpace
| returned status=%08x, ClusterSizeInBytes=%08x,
| TotalClusters=%016l64x\n",status,ClusterSizeInBytes,Tota
| lClusters.QuadPart));
79089|
79090|     if ( NT_SUCCESS(status) ) {
79091|         if ( VirginMap.isEmpty() ) {
79092|             VirginDebug(("TestVirginMap: The VirginMap
| is empty! No overwrite test will be performed.\n"));
79093|         } else {
79094|             status = TestVirginMap_OverwriteAllExtents
| (Volume, ClusterSizeInBytes, TotalClusters, VirginMap);
79095|         }
79096|     }
79097|
79098|     VirginDebug(("Leaving TestVirginMap()\n"));
79099| }
79100| #endif /*DEBUG*/
79101|
79102| //-----
| -----
79103|
79104| /*--- end of file virgin.cpp ---*/

```

```

79105|
79106|
79107|
79108| File Listing: virgin.h
79109|
79110| #ifndef __PSM_VIRGIN_H
79111| #define __PSM_VIRGIN_H 1
79112|
79113| #ifndef __cplusplus
79114|     #error This file must be compiled as C++
79115| #endif /*__cplusplus*/
79116| //-----
79117| | -----
79118| | ---
79119| NTSTATUS GetClusterSize (
79120|     PFILTERED_EXTENSION    DevExt,
79121|     ULONG                   &ClusterSizeInBytes,
79122|     LARGE_INTEGER           &TotalClusters );
79123| //-----
79124| | -----
79125| | ---
79126| #endif /*__PSM_VIRGIN_H*/
79127| /*--- end of file virgin.h ---*/
79128|
79129| File Listing: VM.cpp
79130|
79131| #include "precomp.h"
79132|
79133| #if _WIN32_WINNT < 0x0500
79134| /*
79135|  * void_devfile_open - open a device file given a
79136|  * | pathname
79137|  * |
79138|  * Return a non-zero void error code for error.
79139|  */
79140| NTSTATUS void_devfile_open(HANDLE *Handle, WCHAR
79141|     | *PathName)
79142| {
79143|     OBJECT_ATTRIBUTES    ObjAttrs;
79144|     UNICODE_STRING       UniPath;
79145|     NTSTATUS              Status;
79146|     IO_STATUS_BLOCK       IoStatus;
79147|
79148|     *Handle = NULL;
79149|     RtlInitUnicodeString(&UniPath, PathName);

```

```

79149|   InitializeObjectAttributes(&ObjAttrs, &UniPath,
    | OBJ_CASE_INSENSITIVE, NULL, NULL);
79150|
79151|   Status = ZwOpenFile(Handle,
79152|       SYNCHRONIZE | FILE_READ_DATA |
    | FILE_WRITE_DATA,
79153|       &ObjAttrs, &IoStatus,
79154|       FILE_SHARE_READ |
    | FILE_SHARE_WRITE,
79155|       FILE_SYNCHRONOUS_IO_ALERT);
79156|
79157|   return Status;
79158| }
79159|
79160|
79161| /*
79162|  * void_devfile_ioctl - issue an ioctl to a device
79163|  */
79164| NTSTATUS void_devfile_ioctl(
79165|     HANDLE Handle, ULONG Cmd,
79166|     void *inbuf, ULONG
    | inbufsize,
79167|     void *outbuf, ULONG
    | outbufsize)
79168| {
79169|     NTSTATUS    Status;
79170|     IO_STATUS_BLOCK IoStatus;
79171|
79172|     Status = ZwDeviceIoControlFile(Handle,
79173|         (HANDLE) NULL,
79174|         (PIO_APC_ROUTINE)
    | NULL,
79175|         (void *) NULL,
79176|         &IoStatus,
79177|         Cmd,
79178|         inbuf, inbufsize,
79179|         outbuf, outbufsize);
79180|
79181|     return Status;
79182| }
79183|
79184|
79185|
79186| /*
79187|  * void_driver_ioctl - issue a standard LDM device
    | driver ioctl
79188|  *
79189|  * These ioctls are wrapped in a structure that can be
    | used to get
79190|  * the return value. If the ioctl encounters an error,

```

```

    | then return -1.
79191| * Also, these ioctls take a simple pointer, rather
    | than two pointers
79192| * and two sizes like general NT ioctls.
79193| */
79194| NTSTATUS vold_driver_ioctl(HANDLE Handle, int Cmd, void
    | *arg)
79195| {
79196|     struct volnt_iocarg ntarg;
79197|     NTSTATUS Status;
79198|
79199|     ntarg.ntioc_arg = arg;
79200|     ntarg.ntioc_rval = 0;
79201|
79202|     Status = vold_devfile_ioctl(Handle, (ULONG)Cmd,
    | &ntarg, sizeof (ntarg),
79203|         &ntarg, sizeof (ntarg));
79204|     return Status;
79205| }
79206|
79207| #endif
79208|
79209|
79210|
79211| File Listing: WMI.cpp
79212|
79213| #include "precomp.h"
79214|
79215| #if _WIN32_WINNT >= 0x0500
79216|
79217| STATIC NTSTATUS
79218| PSMANWmiObject(
79219|     IN PDEVICE_OBJECT DeviceObject,
79220|     IN PIRP Irp
79221| );
79222|
79223| STATIC NTSTATUS
79224| PSMANWmiDevice(
79225|     IN PDEVICE_OBJECT DeviceObject,
79226|     IN PIRP Irp
79227| );
79228|
79229| STATIC NTSTATUS PSMANWmiVDisk(
79230|     IN PDEVICE_OBJECT DeviceObject,
79231|     IN PIRP Irp
79232| );
79233| /*-----
    | -----*/
79234| NTSTATUS
79235| PSMANWmi(

```



```

79236| IN PDEVICE_OBJECT DeviceObject,
79237| IN PIRP Irp
79238| )
79239|
79240| /*++
79241|
79242| Routine Description:
79243|
79244|     Passes the Irp to the correct handler
79245|
79246| Arguments:
79247|
79248|     DriverObject - Pointer to device object to being
| shutdown by system.
79249|     Irp          - IRP involved.
79250|
79251| Return Value:
79252|
79253|     NT Status
79254|
79255| --*/
79256|
79257| {
79258|
79259|     NTSTATUS Status;
79260|
79261|     switch(PsmGetObjectType(DeviceObject)) {
79262|     case OBJECT_INTERNAL :
79263|         Status = PSMANWmiObject(DeviceObject, Irp);
79264|         break;
79265|     case OBJECT_FILTEREDDISK :
79266|         Status = PSMANWmiDevice(DeviceObject, Irp);
79267|         break;
79268|     case OBJECT_VIRTUALDISK :
79269|         Status = PSMANWmiVDisk(DeviceObject, Irp);
79270|         break;
79271|     case OBJECT_FS_FILTER :
79272|         Status = PSMANWmiFSFilter(DeviceObject,
| Irp);
79273|         break;
79274|     case OBJECT_FS_OBJECT :
79275|         Status = PSMANWmiFSObject(DeviceObject,
| Irp);
79276|         break;
79277|     default:
79278|         Irp->IoStatus.Status = Status =
| STATUS_NO_SUCH_DEVICE;
79279|         Irp->IoStatus.Information = 0 ;
79280|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
79281|         break;

```

```

79282|    }
79283|    return Status;
79284|
79285| } // end PSMANWmi()
79286|
79287|
79288| /*-----
| -----*/
79289| STATIC NTSTATUS
79290| PSMANWmiObject(
79291|     IN PDEVICE_OBJECT DeviceObject,
79292|     IN PIRP Irp
79293| )
79294|
79295| /*++
79296|
79297| Routine Description:
79298|
79299|     This routine is called for pnp IRPs.
79300|
79301| Arguments:
79302|
79303|     DriverObject - Pointer to device object
79304|     Irp          - IRP involved.
79305|
79306| Return Value:
79307|
79308|     NT Status
79309|
79310| --*/
79311|
79312| {
79313|     NOT_REFERENCED(DeviceObject);
79314|     Debug(DEBUG_PROCCALL,("PSMANWmiObject Called\n"));
79315|     Irp->IoStatus.Status = STATUS_SUCCESS;
79316|     Irp->IoStatus.Information = 0;
79317|
79318|     IoCompleteRequest(Irp, IO_NO_INCREMENT);
79319|     Debug(DEBUG_PROCCALL,("PSMANWmiObject Done\n"));
79320|     return STATUS_SUCCESS;
79321|
79322| } // end PSMANWmiObject()
79323|
79324| NTSTATUS
79325| PSMANQueryWmiRegInfo(
79326|     IN PDEVICE_OBJECT DeviceObject,
79327|     OUT ULONG *RegFlags,
79328|     OUT PUNICODE_STRING InstanceName,
79329|     OUT PUNICODE_STRING *RegistryPath,
79330|     OUT PUNICODE_STRING MofResourceName,

```

```

79331|   OUT PDEVICE_OBJECT *Pdo
79332|   )
79333| /*++
79334|
79335| Routine Description:
79336|
79337|   This routine is a callback into the driver to
79338|   | retrieve information about
79339|   the guids being registered.
79340|
79341| Implementations of this routine may be in paged
79342| | memory
79343|
79344| Arguments:
79345|
79346|   DeviceObject is the device whose registration
79347|   | information is needed
79348|
79349|   *RegFlags returns with a set of flags that describe
79350|   | all of the guids being
79351|   registered for this device. If the device wants
79352|   | enable and disable
79353|   collection callbacks before receiving queries
79354|   | for the registered
79355|   guids then it should return the
79356|   | WMIREG_FLAG_EXPENSIVE flag. Also the
79357|   returned flags may specify
79358|   | WMIREG_FLAG_INSTANCE_PDO in which case
79359|   the instance name is determined from the PDO
79360|   | associated with the
79361|   device object. Note that the PDO must have an
79362|   | associated devnode. If
79363|   WMIREG_FLAG_INSTANCE_PDO is not set then Name
79364|   | must return a unique
79365|   name for the device. These flags are ORed into
79366|   | the flags specified
79367|   by the GUIDREGINFO for each guid.
79368|
79369| InstanceName returns with the instance name for the
79370| | guids if
79371| WMIREG_FLAG_INSTANCE_PDO is not set in the
79372| | returned *RegFlags. The
79373| caller will call ExFreePool with the buffer
79374| | returned.
79375|
79376|
79377| *RegistryPath returns with the registry path of the
79378| | driver. This is
79379| required
79380|
79381|
79382| MofResourceName returns with the name of the MOF

```

```

    | resource attached to
79365|     the binary file. If the driver does not have a
    | mof resource attached
79366|     then this can be returned unmodified. If a
    | value is returned then
79367|     it is NOT freed.
79368|
79369| *Pdo returns with the device object for the PDO
    | associated with this
79370|     device if the WMIREG_FLAG_INSTANCE_PDO flag is
    | returned in
79371|     *RegFlags.
79372|
79373| Return Value:
79374|
79375|     status
79376|
79377| --*/
79378| {
79379|     USHORT size;
79380|     NTSTATUS status;
79381|     PFILTERED_EXTENSION deviceExtension =
        | (PFILTERED_EXTENSION)GetDeviceExtension(DeviceObject);
79382|
79383|     PAGED_CODE();
79384|
79385|     size = deviceExtension->PhysicalDeviceName.Length +
        | sizeof(UNICODE_NULL);
79386|
79387|     InstanceName->Buffer = (WCHAR
        | *)ExAllocatePool(PagedPool, size);
79388|     if (InstanceName->Buffer != NULL) {
79389|         *RegistryPath = &gRegistryPath;
79390|
79391|         *RegFlags = WMIREG_FLAG_INSTANCE_PDO |
            | WMIREG_FLAG_EXPENSIVE;
79392|         *Pdo = deviceExtension->PhysicalDeviceObject;
79393|         status = STATUS_SUCCESS;
79394|     } else {
79395|         status = STATUS_INSUFFICIENT_RESOURCES;
79396|     }
79397|
79398|     return(status);
79399| }
79400|
79401|
79402| NTSTATUS
79403| PSMQueryWmiDataBlock(
79404|     IN PDEVICE_OBJECT DeviceObject,
79405|     IN PIRP Irp,

```

```

79406|    IN ULONG GuidIndex,
79407|    IN ULONG InstanceIndex,
79408|    IN ULONG InstanceCount,
79409|    IN OUT PULONG InstanceLengthArray,
79410|    IN ULONG BufferAvail,
79411|    OUT PCHAR Buffer
79412|    )
79413| /*++
79414|
79415| Routine Description:
79416|
79417|    This routine is a callback into the driver to query
79418|    | for the contents of
79419|    all instances of a data block. When the driver has
79420|    | finished filling the
79421|    data block it must call WmiCompleteRequest to
79422|    | complete the irp. The
79423|    driver can return STATUS_PENDING if the irp cannot
79424|    | be completed
79425|    immediately.
79426|
79427| Arguments:
79428|
79429|    DeviceObject is the device whose data block is
79430|    | being queried
79431|
79432|    Irp is the Irp that makes this request
79433|
79434|    GuidIndex is the index into the list of guides
79435|    | provided when the
79436|    device registered
79437|
79438|    InstanceCount is the number of instances expected
79439|    | to be returned for
79440|    the data block.
79441|
79442|    InstanceLengthArray is a pointer to an array of
79443|    | ULONG that returns the
79444|    lengths of each instance of the data block. If
79445|    | this is NULL then
79446|    there was not enough space in the output buffer
79447|    | to fulfill the request
79448|    so the irp should be completed with the buffer
79449|    | needed.
79450|
79451|    BufferAvail on entry has the maximum size available
79452|    | to write the data
79453|    blocks.
79454|
79455|    Buffer on return is filled with the returned data

```

```

| blocks. Note that each
79444| instance of the data block must be aligned on a
| 8 byte boundry.
79445|
79446|
79447| Return Value:
79448|
79449| status
79450|
79451| --*/
79452| {
79453| NTSTATUS status;
79454| PFILTERED_EXTENSION deviceExtension;
79455| ULONG sizeNeeded;
79456| PDISK_PERFORMANCE totalCounters;
79457| PDISK_PERFORMANCE diskCounters;
79458| PWMI_DISK_PERFORMANCE diskPerformance;
79459| ULONG deviceNameSize;
79460| PWCHAR diskNamePtr;
79461|
79462| deviceExtension = (PFILTERED_EXTENSION)
| GetDeviceExtension(DeviceObject);
79463|
79464| if (GuidIndex == 0) {
79465| deviceNameSize =
| deviceExtension->PhysicalDeviceName.Length +
| sizeof(USHORT);
79466| sizeNeeded = ((sizeof(WMI_DISK_PERFORMANCE) +
| 1) & ~1) + deviceNameSize;
79467| diskCounters = deviceExtension->DiskCounters;
79468| if (diskCounters == NULL) {
79469| status = STATUS_UNSUCCESSFUL;
79470| } else
79471| if (BufferAvail >= sizeNeeded) {
79472| //
79473| // Update idle time if disk has been idle
79474| //
79475| ULONG i;
79476|
79477| RtlZeroMemory(Buffer,
| sizeof(WMI_DISK_PERFORMANCE));
79478| diskPerformance =
| (PWMI_DISK_PERFORMANCE)Buffer;
79479|
79480| totalCounters =
| (PDISK_PERFORMANCE)diskPerformance;
79481|
79482| for (i=0; i<deviceExtension->Processors;
| i++) {
79483| PSManAddCounters( totalCounters,

```

```

    | diskCounters);
79484|         diskCounters = (PDISK_PERFORMANCE)
    | ((PCHAR)diskCounters + PROCESSOR_COUNTERS_SIZE);
79485|     }
79486|     totalCounters->QueueDepth =
    | deviceExtension->QueueDepth;
79487|
    | KeQuerySystemTime(&totalCounters->QueryTime);
79488|
79489|     if (totalCounters->QueueDepth == 0) {
79490|         LARGE_INTEGER difference;
79491|
79492|         difference.QuadPart =
    | totalCounters->QueryTime.QuadPart -
    | deviceExtension->LastIdleClock.QuadPart;
79493|         totalCounters->IdleTime.QuadPart +=
    | difference.QuadPart;
79494|     }
79495|     totalCounters->StorageDeviceNumber =
    | deviceExtension->DiskNumber;
79496|     RtlCopyMemory(
    | &totalCounters->StorageManagerName[0],
    | &deviceExtension->StorageManagerName[0], 8 *
    | sizeof(WCHAR));
79497|
79498|     diskNamePtr = (PWCHAR)(Buffer +
    | ((sizeof(DISK_PERFORMANCE) + 1) & ~1));
79499|     *diskNamePtr++ =
    | deviceExtension->PhysicalDeviceName.Length;
79500|     RtlCopyMemory(diskNamePtr,
    | deviceExtension->PhysicalDeviceName.Buffer,
    | deviceExtension->PhysicalDeviceName.Length);
79501|     *InstanceLengthArray = sizeNeeded;
79502|
79503|     status = STATUS_SUCCESS;
79504| } else {
79505|     status = STATUS_BUFFER_TOO_SMALL;
79506| }
79507|
79508| } else {
79509|     status = STATUS_WMI_GUID_NOT_FOUND;
79510| }
79511|
79512| status = WmiCompleteRequest(
    | DeviceObject,Irp,status,sizeNeeded,IO_NO_INCREMENT);
79513| return(status);
79514| }
79515|
79516|
79517| NTSTATUS

```

```

79518| PSMANWmiFunctionControl(
79519|     IN PDEVICE_OBJECT DeviceObject,
79520|     IN PIRP Irp,
79521|     IN ULONG GuidIndex,
79522|     IN WMIENABLEDISABLECONTROL Function,
79523|     IN BOOLEAN Enable
79524| )
79525| /*++
79526|
79527| Routine Description:
79528|
79529|     This routine is a callback into the driver to query
79530|     | for enabling or
79531|     disabling events and data collection. When the
79532|     | driver has finished it
79533|     must call WmiCompleteRequest to complete the irp.
79534|     | The driver can return
79535|     STATUS_PENDING if the irp cannot be completed
79536|     | immediately.
79537|
79538| Arguments:
79539|
79540|     DeviceObject is the device whose events or data
79541|     | collection are being
79542|     enabled or disabled
79543|
79544|     Irp is the Irp that makes this request
79545|
79546|     GuidIndex is the index into the list of guides
79547|     | provided when the
79548|     device registered
79549|
79550|     Function differentiates between event and data
79551|     | collection operations
79552|
79553|     Enable indicates whether to enable or disable
79554|
79555| Return Value:
79556|
79557|     status
79558|
79559| --*/
79560| {
79561|     NTSTATUS status;
79562|     PFILTERED_EXTENSION deviceExtension;
79563|
79564|     deviceExtension = (PFILTERED_EXTENSION)
79565|     | GetDeviceExtension(DeviceObject);
79566|
79567|

```



```

79560|   if (GuidIndex == 0) {
79561|       if (Function == WmiDataBlockControl) {
79562|           if (Enable) {
79563|               if
79564|               | (InterlockedIncrement((PLONG)&deviceExtension->CountersE
79565|               | nabled) == 1) {
79566|                   //
79567|                   // Reset per processor counters to 0
79568|                   //
79569|                   if (deviceExtension->DiskCounters !=
79570|                   | NULL) {
79571|                       | RtlZeroMemory(deviceExtension->DiskCounters,PROCESSOR_CO
79572|                       | UNTERS_SIZE * deviceExtension->Processors);
79573|                   }
79574|               }
79575|           } else {
79576|               if
79577|               | (InterlockedDecrement((PLONG)&deviceExtension->CountersE
79578|               | nabled)<= 0) {
79579|                   deviceExtension->CountersEnabled = 0;
79580|                   deviceExtension->QueueDepth = 0;
79581|                   Debug(DEBUG_WMI,("PSManWmi: Counters
79582|                   | disabled %d\n",deviceExtension->CountersEnabled));
79583|               }
79584|           }
79585|       }
79586|   }
79587|   status =
79588|   | WmiCompleteRequest(DeviceObject,Irp,status,0,IO_NO_INCRE
79589|   | MENT);
79590|   return(status);
79591| }
79592| /*-----*/
79593| | -----*/
79594| STATIC NTSTATUS
79595| PSMANWMI_DEVICE(
79596|     IN PDEVICE_OBJECT DeviceObject,
79597|     IN PIRP Irp

```

```

79597|    )
79598|
79599| /*++
79600|
79601| Routine Description:
79602|
79603|    This routine handles any WMI requests for
79604|    information. Since the disk
79605|    information is read-only, is always collected and
79606|    does not have any
79607|    events only QueryAllData, QuerySingleInstance and
79608|    GetRegInfo requests
79609|    are supported.
79610|
79611| Arguments:
79612|
79613|    DeviceObject - Context for the activity.
79614|    Irp          - The device control argument block.
79615|
79616| Return Value:
79617|
79618|    Status is returned.
79619|
79620| --*/
79621| {
79622|    PIO_STACK_LOCATION
79623|    | IrpSp=IoGetCurrentIrpStackLocation(Irp);
79624|    NTSTATUS status;
79625|    PWMI_LIB_CONTEXT WmiLibContext;
79626|    SYSCTL_IRP_DISPOSITION disposition;
79627|    PFILTERED_EXTENSION deviceExtension =
79628|    | (PFILTERED_EXTENSION)GetDeviceExtension(DeviceObject);
79629|    PAGED_CODE();
79630|
79631|    Debug(DEBUG_WMI,( "PSManWmi: DeviceObject %X Irp %X
79632|    | %d - %s\n",DeviceObject,
79633|    | Irp,IrpSp->MinorFunction,File_GetSystemControlMinorFunct
79634|    | ionName(IrpSp->MinorFunction)));
79635|    WmiLibContext = &deviceExtension->WmiLibContext;
79636|    if (WmiLibContext->GuidCount == 0) { //
79637|    | WmiLibContext is not valid
79638|
79639|    Debug(DEBUG_WMI,( "PSManWmi: WmiLibContext
79640|    | invalid\n"));
79641|
79642|    return PSManPassThru(DeviceObject, Irp);
79643|    }
79644| #if 0
79645|    if (IrpSp->MinorFunction ==
79646|    | IRP_MN_SET_TRACE_NOTIFY) {

```

```

79636|     PVOID buffer = irpSp->Parameters.WMI.Buffer;
79637|     ULONG bufferSize =
79638|         | irpSp->Parameters.WMI.BufferSize;
79639|     if (bufferSize <
79640|         | sizeof(PPHYSICAL_DISK_IO_NOTIFY_ROUTINE)) {
79641|         status = STATUS_BUFFER_TOO_SMALL;
79642|     } else {
79643|         //
79644|         // First we need to turn on counters if we
79645|         | are doing tracing
79646|         //
79647|         PVOID current, notifyRoutine;
79648|         ULONG i;
79649|         current = (PVOID)
79650|             | deviceExtension->PhysicalDiskIoNotifyRoutine;
79651|         notifyRoutine = *((PVOID *)buffer);
79652|         if (current == NULL && notifyRoutine !=
79653|             | NULL) {
79654|             if
79655|                 | (InterlockedIncrement(&deviceExtension->CountersEnabled)
79656|                 | == 1) {
79657|                 //
79658|                 // reset per processor counters
79659|                 | only
79660|                 //
79661|                 if (deviceExtension->DiskCounters
79662|                     | != NULL) {
79663|                     RtlZeroMemory(
79664|                         | deviceExtension->DiskCounters, PROCESSOR_COUNTERS_SIZE
79665|                         | * deviceExtension->Processors);
79666|                     }
79667|                     | KeQuerySystemTime(&deviceExtension->LastIdleClock);
79668|                     deviceExtension->QueueDepth = 0;
79669|                 }
79670|                 Debug(DEBUG_WMI, ("PSManWmi: Counters
79671|                     | enabled %d\n", deviceExtension->CountersEnabled));
79672|             } else
79673|                 if (current != NULL && notifyRoutine ==
79674|                     | NULL) {
79675|                     | InterlockedDecrement(&deviceExtension->CountersEnabled);
79676|                     deviceExtension->QueueDepth = 0;
79677|                     Debug(DEBUG_WMI, ("PSManWmi: Counters
79678|                         | disabled %d\n", deviceExtension->CountersEnabled));
79679|                 }
79680|                 | deviceExtension->PhysicalDiskIoNotifyRoutine =

```

```

    | (PPHYSICAL_DISK_IO_NOTIFY_ROUTINE)*((PVOID *)buffer);
79669|
79670|     Debug(DEBUG_WMI,("PSManWmi:
    | SET_TRACE_NOTIFY to
    | %X\n",deviceExtension->PhysicalDiskIoNotifyRoutine));
79671|     status = STATUS_SUCCESS;
79672| }
79673|
79674|     Irp->IoStatus.Status = status;
79675|     Irp->IoStatus.Information = 0;
79676|     IoCompleteRequest( Irp, IO_NO_INCREMENT );
79677| } else {
79678|     Debug(DEBUG_WMI,( "PSManWmi: Calling
    | WmiSystemControl\n"));
79679|     status =
    | WmiSystemControl(wmilibContext,DeviceObject,Irp,&disposi
    | tion);
79680|     switch (disposition) {
79681|         case IrpProcessed: {
79682|             break;
79683|         }
79684|         case IrpNotCompleted: {
79685|             IoCompleteRequest(Irp,
    | IO_NO_INCREMENT);
79686|             break;
79687|         }
79688|
79689| //         case IrpForward:
79690| //         case IrpNotWmi:
79691|         default: {
79692|             status = PSManPassThru(DeviceObject,
    | Irp);
79693|             break;
79694|         }
79695|     }
79696| }
79697| #else
79698| // changed 3/7/2000 above doesnt work with release
    | of Win2000 DDK
79699|     Debug(DEBUG_WMI,( "PSManWmi: Calling
    | WmiSystemControl\n"));
79700|     status =
    | WmiSystemControl(wmilibContext,DeviceObject,Irp,&disposi
    | tion);
79701|     switch (disposition) {
79702|         case IrpProcessed: {
79703|             break;
79704|         }
79705|         case IrpNotCompleted: {
79706|             IoCompleteRequest(Irp, IO_NO_INCREMENT);

```

```

79707|         break;
79708|     }
79709|
79710| //         case IrpForward:
79711| //         case IrpNotWmi:
79712|     default: {
79713|         status = PSMANPassThru(DeviceObject, Irp);
79714|         break;
79715|     }
79716| }
79717|
79718| #endif
79719|     return(status);
79720|
79721| } // end PSMANWmiDevice()
79722|
79723|
79724| /*-----
    | -----*/
79725| STATIC NTSTATUS PSMANWmiVDisk(
79726|     IN PDEVICE_OBJECT DeviceObject,
79727|     IN PIRP Irp
79728| )
79729| {
79730|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
79731|
79732|     Debug(DEBUG_PROCCALL | DEBUG_WMI,("PSMANWmiVDisk
    | Called Dev=%p, Irp=%p\n", DeviceObject, Irp));
79733|     Irp->IoStatus.Information = 0;
79734|     Irp->IoStatus.Status = Status;
79735|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
79736|     Debug(DEBUG_PROCCALL | DEBUG_WMI,("PSMANWmiVDisk
    | Done\n"));
79737|
79738|     return Status;
79739| }
79740|
79741| /*-----
    | -----*/
79742| STATIC NTSTATUS PSMANWmiFSObject(
79743|     IN PDEVICE_OBJECT DeviceObject,
79744|     IN PIRP Irp
79745| )
79746| {
79747|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
79748|
79749|     Debug(DEBUG_PROCCALL | DEBUG_WMI,("PSMANWmiFSObject
    | Called Dev=%p, Irp=%p\n", DeviceObject, Irp));
79750|     Irp->IoStatus.Information = 0;
79751|     Irp->IoStatus.Status = Status;

```

```

79752| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
79753| Debug(DEBUG_PROCCALL | DEBUG_WMI,("PSManWmiFSObject
| Done\n"));
79754|
79755| return Status;
79756| }
79757|
79758|
79759| /*-----
| -----*/
79760| STATIC NTSTATUS
79761| PSManWmiFSFilter(
79762|     IN PDEVICE_OBJECT DeviceObject,
79763|     IN PIRP Irp
79764| )
79765|
79766| /*++
79767|
79768| Routine Description:
79769|
79770|     Pass irp to handler
79771|
79772| Arguments:
79773|
79774|     DriverObject - Pointer to device object to being
| shutdown by system.
79775|     Irp          - IRP involved.
79776|
79777| Return Value:
79778|
79779|     NT Status
79780|
79781| --*/
79782|
79783| {
79784|     NTSTATUS Status;
79785|
79786| #ifdef DEBUG
79787|     if(PsmActive) {
79788|         Debug(DEBUG_WMI |
| DEBUG_PROCCALL,("PSManWmiFSFilter Called Device=%p,
| Irp=%p\n",DeviceObject,Irp));
79789|     }
79790| #endif
79791|
79792|     Status = PSManFSPassThru( DeviceObject, Irp );
79793|
79794| #ifdef DEBUG
79795|     if(PsmActive) {
79796|         Debug(DEBUG_WMI |

```

```

    | DEBUG_PROCCALL,("PSManWmiFSFilter Done Device=%p,
    | Irp=%p, Status=%08x\n",DeviceObject,Irp,Status));
79797|    }
79798| #endif
79799|    return Status;
79800| } // end PSManWmiFSFilter()
79801|
79802|
79803|
79804| #endif
79805|
79806|
79807|
79808| File Listing: WMI.h
79809|
79810| NTSTATUS
79811| PSManWmi(
79812|     IN PDEVICE_OBJECT DeviceObject,
79813|     IN PIRP Irp
79814| );
79815| NTSTATUS
79816| PSManQueryWmiDataBlock(
79817|     IN PDEVICE_OBJECT DeviceObject,
79818|     IN PIRP Irp,
79819|     IN ULONG GuidIndex,
79820|     IN ULONG InstanceIndex,
79821|     IN ULONG InstanceCount,
79822|     IN OUT PULONG InstanceLengthArray,
79823|     IN ULONG BufferAvail,
79824|     OUT PCHAR Buffer
79825| );
79826| NTSTATUS
79827| PSManWmiFunctionControl(
79828|     IN PDEVICE_OBJECT DeviceObject,
79829|     IN PIRP Irp,
79830|     IN ULONG GuidIndex,
79831|     IN WMIENABLEDISABLECONTROL Function,
79832|     IN BOOLEAN Enable
79833| );
79834| NTSTATUS
79835| PSManQueryWmiRegInfo(
79836|     IN PDEVICE_OBJECT DeviceObject,
79837|     OUT ULONG *RegFlags,
79838|     OUT PUNICODE_STRING InstanceName,
79839|     OUT PUNICODE_STRING *RegistryPath,
79840|     OUT PUNICODE_STRING MofResourceName,
79841|     OUT PDEVICE_OBJECT *Pdo
79842| );
79843| NTSTATUS
79844| PSManWmiFSObject(

```

```

79845|    IN PDEVICE_OBJECT DeviceObject,
79846|    IN PIRP Irp
79847|    );
79848| NTSTATUS
79849| PSMANWmiFSFilter(
79850|    IN PDEVICE_OBJECT DeviceObject,
79851|    IN PIRP Irp
79852|    );
79853|
79854|
79855|
79856| File Listing: WRITE.cpp
79857|
79858| #include "precomp.h"
79859|
79860| // note: define DO_ALL_IO in precomp.h do get all io
    | related functions
79861| // otherwise only io in this file will get printed
79862|
79863| // define to not use a write cache.
79864| //#define _NO_WRITE_ROUTINES_ 1
79865|
79866| // define if writes should return error on no media
    | loaded, or "Okay" if not defined
79867| // should be set to 0 so we do not get "Lost delayed
    | write" popups on server
79868| #define DO_ERROR_WRITES 1
79869|
79870| /*lint -save -e767*/
79871| // Try and keep the "lost delayed write" messages from
    | popping up
79872| #if DO_ERROR_WRITES
79873|    #define VOLUME_NOT_ACTIVE_ERROR_CODE
        | STATUS_NO_MEDIA_IN_DEVICE
        | /*STATUS_DEVICE_NOT_CONNECTED*/
79874|    #define INFORMATION_FOR_NOT_ACTIVE(Irp) 0
79875| #else
79876|    #define VOLUME_NOT_ACTIVE_ERROR_CODE STATUS_SUCCESS
79877|    #define INFORMATION_FOR_NOT_ACTIVE(Irp)
        | IoGetCurrentIrpStackLocation(Irp)->Parameters.Write.Leng
        | th
79878| #endif
79879| /*lint -restore*/
79880|
79881| // set to 0 if you need to mount a volume with no
    | virtual writes on
79882| // it as we need to in the rebuild process for
    | freespace
79883| ULONG gVDiskDoVirtualIO=TRUE;
79884|

```



```

79885| STATIC NTSTATUS AddSectorsToMemoryCache(
79886|                                     PVDISK_EXTENSION
79887|                                     | DevExt,
79888|                                     | Sector,
79889|                                     | Count,
79890|                                     | *Buffer );
79891|
79892| /*-----
79893| | -----*/
79894| NTSTATUS
79895| PSMANWrite(
79896|     IN PDEVICE_OBJECT DeviceObject,
79897|     IN IRP Irp
79898| )
79899| /*++
79900|
79901| Routine Description:
79902|
79903| This is the driver entry point for write requests
79904| to disks to which the PSMAN driver has attached.
79905| This driver collects statistics and then sets a
79906| completion
79907| routine so that it can collect additional
79908| information when
79909| the request completes. Then it calls the next
79910| driver below
79911| it.
79912|
79913| Arguments:
79914|
79915| DeviceObject
79916| Irp
79917|
79918| Return Value:
79919| NTSTATUS
79920|
79921| --*/
79922| {
79923|     switch ( PsmGetObjectTypes(DeviceObject) ) {
79924|     case OBJECT_INTERNAL :
79925|         return PSMANWriteObject(DeviceObject, Irp);
79926|     case OBJECT_FILTEREDDISK :
79927|         return PSMANWriteDevice(DeviceObject, Irp);

```

```

79927|     case OBJECT_VIRTUALDISK :
79928|         return PSMANWriteVdisk(DeviceObject, Irp);
79929|     case OBJECT_FS_OBJECT :
79930|         return PSMANWriteFSObject(DeviceObject,
79931|             | Irp);
79932|     case OBJECT_FS_FILTER :
79933|         return PSMANWriteFSFilter(DeviceObject,
79934|             | Irp);
79935|     default:
79936|         Irp->IoStatus.Status =
79937|             | STATUS_NO_SUCH_DEVICE;
79938|         Irp->IoStatus.Information = 0 ;
79939|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
79940|         return STATUS_NO_SUCH_DEVICE;
79941|     }
79942| } // PSMANWrite
79943|
79944| typedef struct sFilesToSkip {
79945|     PFILE_OBJECT FileObject;
79946|     LIST_ENTRY ListEntry;
79947| } tFilesToSkip, *pFilesToSkip;
79948|
79949| STATIC LIST_ENTRY FilesToSkip;
79950| STATIC KSPIN_LOCK FilesToSkipSpinLock;
79951|
79952| NTSTATUS InitWriteModule( )
79953| {
79954|     InitializeListHead(&FilesToSkip);
79955|     KeInitializeSpinLock(&FilesToSkipSpinLock);
79956|     | pmRegisterObject(&FilesToSkipSpinLock,"FilesToSkipSpinLo
79957|     | ck",pmSpinLock);
79958|     return STATUS_SUCCESS;
79959| }
79960|
79961| NTSTATUS InsertFileObjectToSkip ( PFILE_OBJECT
79962|     | FileObject )
79963| {
79964|     KIRQL _oldIrql_;
79965|     tFilesToSkip *Skip;
79966|
79967|     | Skip=(tFilesToSkip*)MemAllocatePoolWithTag(NonPagedPool,
79968|     | sizeof(tFilesToSkip),PSM_SKIP_FILE_TAG);
79969|     if ( Skip ) {
79970|         Skip->FileObject = FileObject;
79971|         pmAcquireSpinLock ( &FilesToSkipSpinLock,
79972|             | &_oldIrql_ );
79973|         InsertTailList(&FilesToSkip,&Skip->ListEntry);

```

```

79968|     pmReleaseSpinLock( &FilesToSkipSpinLock,
    | _oldIrql_ );
79969|     return STATUS_SUCCESS;
79970| } else {
79971|     return STATUS_INSUFFICIENT_RESOURCES;
79972| }
79973| }
79974|
79975| NTSTATUS DeleteFileObjectToSkip ( PFILE_OBJECT
    | FileObject )
79976| {
79977|     KIRQL _oldIrql_;
79978|     tFilesToSkip *Skip;
79979|     PLIST_ENTRY ListEntry;
79980|
79981|     pmAcquireSpinLock ( &FilesToSkipSpinLock,
    | &_oldIrql_ );
79982|     ListEntry = FilesToSkip.Flink;
79983|
79984|     while ( ListEntry!=&FilesToSkip ) {
79985|         Skip =
    | CONTAINING_RECORD(ListEntry,tFilesToSkip,ListEntry);
79986|         if ( Skip->FileObject == FileObject ) {
79987|             RemoveEntryList(ListEntry);
79988|             MemFreePool(Skip);
79989|             KeReleaseSpinLock ( &FilesToSkipSpinLock,
    | _oldIrql_ );
79990|             return STATUS_SUCCESS;
79991|         }
79992|         ListEntry = ListEntry->Flink;
79993|     }
79994|
79995|     pmReleaseSpinLock( &FilesToSkipSpinLock, _oldIrql_
    | );
79996|     return STATUS_NOT_FOUND;
79997| }
79998|
79999| NTSTATUS IsFileObjectToSkip ( PFILE_OBJECT FileObject )
80000| {
80001|     KIRQL _oldIrql_;
80002|     tFilesToSkip *Skip;
80003|     PLIST_ENTRY ListEntry;
80004|
80005|     pmAcquireSpinLock ( &FilesToSkipSpinLock,
    | &_oldIrql_ );
80006|     ListEntry = FilesToSkip.Flink;
80007|
80008|     while ( ListEntry!=&FilesToSkip ) {
80009|         Skip =
    | CONTAINING_RECORD(ListEntry,tFilesToSkip,ListEntry);

```

```

80010|     if ( Skip->FileObject == FileObject ) {
80011|         KeReleaseSpinLock ( &FilesToSkipSpinLock,
80012|             | _oldIrql_);
80013|         return STATUS_SUCCESS;
80014|     }
80015|     ListEntry = ListEntry->Flink;
80016| }
80017| pmReleaseSpinLock( &FilesToSkipSpinLock, _oldIrql_
80018|     | );
80019| return STATUS_NOT_FOUND;
80020| }
80021|
80022|
80023| /*-----
80024| | -----*/
80025| STATIC NTSTATUS
80026| PSMANWriteObject(
80027|     IN PDEVICE_OBJECT DeviceObject,
80028|     IN PIRP Irp
80029| )
80030| /*++
80031|
80032| Routine Description:
80033|
80034| This is the driver entry point for write requests
80035| to disks to which the PSMAN driver has attached.
80036| This driver collects statistics and then sets a
80037| completion
80038| routine so that it can collect additional
80039| information when
80040| the request completes. Then it calls the next
80041| driver below
80042| it.
80043|
80044| Arguments:
80045|
80046| DeviceObject
80047| Irp
80048|
80049| Return Value:
80050|
80051| NTSTATUS
80052|
80053| --*/
80054| {
80055|     NTSTATUS Status = STATUS_INVALID_PARAMETER;

```

```

80054| NOT_REFERENCED(DeviceObject);
80055|
80056| Debug(DEBUG_PROCCALL,("PSManWriteObject
    | Called\n"));
80057| Irp->IoStatus.Status = Status;
80058| Irp->IoStatus.Information = 0;
80059|
80060| IoCompleteRequest(Irp, IO_NO_INCREMENT);
80061| Debug(DEBUG_PROCCALL,("PSManWriteObject Done\n"));
80062| return Status;
80063| } // PSManWriteObject
80064|
80065| #if 0
80066| STATIC ULONG PsmNeedToPsm( PFILTERED_EXTENSION DevExt,
    | ULONG Sector, ULONG Count)
80067| {
80068|     ULONG Need=0;
80069|     ULONG j;
80070|
80071|     GetDevExtForRead(DevExt);
80072|     if ( DevExt->PSMSectors ) {
80073|         for ( j=0;j<Count;j++ ) {
80074|             if (
                | RtlCheckBit(DevExt->PSMSectors,Sector+j) ) {
80075|                 Need++;
80076|             }
80077|         }
80078|     } else {
80079|         // need everything if no list
80080|         Need = Count;
80081|     }
80082|
80083|     ReleaseDevExtForRead(DevExt);
80084|     return Need;
80085| }
80086| #endif
80087|
80088| BOOLEAN IsCacheFile( PDEVICE_OBJECT DeviceObject, PIRP
    | Irp )
80089| {
80090|     if (
        | IsFileObjectToSkip(Irp->Tail.Overlay.OriginalFileObject)
        | ==STATUS_SUCCESS ) {
80091|         return TRUE;
80092|     }
80093|
80094|     // if master Irp is for cache file then dont PSM
    | it.
80095|     if ( Irp->Flags & IRP_ASSOCIATED_IRP ) {
80096|         if (

```

```

    | IsFileObjectToSkip(Irp->AssociatedIrp.MasterIrp->Tail.Ov
    | erlay.OriginalFileObject)==STATUS_SUCCESS ) {
80097|         return TRUE;
80098|     }
80099| }
80100| return FALSE;
80101| }
80102|
80103|
80104| /*-----
    | -----*/
80105| STATIC NTSTATUS
80106| PSMANWriteDevice(
80107|     IN PDEVICE_OBJECT DeviceObject,
80108|     IN PIRP Irp
80109| )
80110|
80111| /*++
80112|
80113| Routine Description:
80114|
80115| This is the driver entry point for write requests
80116| to disks to which the PSMAN driver has attached.
80117| This driver collects statistics and then sets a
    | completion
80118| routine so that it can collect additional
    | information when
80119| the request completes. Then it calls the next
    | driver below
80120| it.
80121|
80122| Called at <DISPATCH_LEVEL (PASSIVE_LEVEL or
    | APC_LEVEL)
80123|
80124| Arguments:
80125|
80126| DeviceObject
80127| Irp
80128|
80129| Return Value:
80130|
80131| NTSTATUS
80132|
80133| --*/
80134|
80135| {
80136|     PFILTERED_EXTENSION DevExt =
    | (PFILTERED_EXTENSION)GetDeviceExtension(DeviceObject);
80137|     PIO_STACK_LOCATION currentIrpStack =
    | IoGetCurrentIrpStackLocation(Irp);

```

```

80138| //   PIO_STACK_LOCATION nextIrpStack =
      | IoGetNextIrpStackLocation(Irp);
80139|   NTSTATUS Status=STATUS_PENDING;
80140|   KIRQL oldIrq;
80141|   tWriteRequest *WriteRequest=NULL;
80142|
80143|   ASSERT(DevExt->ObjectType==OBJECT_FILTEREDDISK);
80144| #ifdef DEBUG
80145|   if ( KeGetCurrentIrq() >= DISPATCH_LEVEL ) {
80146|       Debug(DEBUG_WRITE,("PSManWriteDevice: Called at
      | >=DISPATCH_LEVEL\n"));
80147|       DbgBreakPoint();
80148|   }
80149| #endif
80150| #if 0
80151|   // verify we are at passive or apc
80152|   PAGED_CODE();
80153|
80154|   if ( KeGetCurrentIrq() >= DISPATCH_LEVEL ) {
80155|       DbgPrint("PSMan: Going to Bug Check! DO=%p,
      | Irp=%p\n",DeviceObject,Irp);
80156|       | PSManBugCheck(SB_BUG_WRITE_FILE,SB_BUG_ATDISPATCH,KeGetC
      | urrentIrq(),(ULONG)DeviceObject,(ULONG)Irp);
80157|   }
80158| #endif
80159|
80160|
80161|   __try {
80162|
80163|       // tell psm that we have an io. must be
      | <DISPATCH_LEVEL
80164|       GetGlobalDeviceForRead();
80165|
80166|       // if not capturing any devices, going ahead
      | and just send
80167|       // it through. im not going to check to see if
      | the physical disk
80168|       // is being psmmed, as we dont currently allow
      | ONLY the physical disk to be psmmed.
80169|       if ( !DevExt->PSMed ) {
80170|           pmAcquireSpinLock (
      | &DevExt->StatisticsSpinLock, &oldIrq );
80171|           // update the logical partition
80172|           DevExt->SectorsWritten +=
      | (currentIrpStack->Parameters.Write.Length /
      | DevExt->BytesPerSector);
80173|           DevExt->NumberOfWriteRequests++;
80174|
80175|           // update the physical drive

```

```

80176|         //PhyExt->SectorsWritten +=
      | (currentIrpStack->Parameters.Write.Length /
      | PhyExt->BytesPerSector);
80177|         //PhyExt->NumberOfWriteRequests++;
80178|         pmReleaseSpinLock (
      | &DevExt->StatisticsSpinLock, oldIrql );
80179|
80180|         if ( DevExt->SignalWrite ) {
80181|             // inform about write.
80182|
      | KeSetEvent(&(DevExt->WriteEvent),(KPRIORITY)0,FALSE);
80183|         }
80184| #if 0
80185| //7-23-99 Dont do this as if a volume not being psmmed
      | is busy, then we will
80186| //     not be able to get our quiescent period
80187|         if ( PhyExt->SignalWrite ) {
80188|             // inform about write.
80189|
      | KeSetEvent(&(PhyExt->WriteEvent),(KPRIORITY)0,FALSE);
80190|         }
80191| #endif
80192|
80193|         ReleaseGlobalDeviceForRead();
80194|         try_return ( Status = PSMANPassThru(
      | DeviceObject, Irp ) );
80195|     }
80196|
80197|     TRACE( TRACE_WRITE,
80198|           0,
80199|
      | (long)(currentIrpStack->Parameters.Write.ByteOffset.Quad
      | Part / 512),
80200|           currentIrpStack->Parameters.Write.Length
      | / 512,
80201|           currentIrpStack->Parameters.Write.Key,
80202|           "");
80203|
80204|     InterlockedIncrement(
      | (PLONG)&OutstandingRequests );
80205|
80206|     // inform about write.
80207|     if ( DevExt->SignalWrite ) {
80208|         // inform about write.
80209|
      | KeSetEvent(&(DevExt->WriteEvent),(KPRIORITY)0,FALSE);
80210|     }
80211|
80212|     | ASSERT((currentIrpStack->Parameters.Write.Length %

```



```

    | DevExt->BytesPerSector)==0);
80213|
80214|     if ( IsCacheFile(DeviceObject,Irp)) {
80215|         #if DO_ALL_SEARCH
80216|             File_PrintOneLiner("Skip
    | File",DeviceObject,Irp);
80217|         #endif
80218|         goto SendOrigNoAudit;
80219|     }
80220|
80221|     if ( (!DoPagingFile) &&
    | (File_IsPagingFile(DeviceObject,Irp)) ) {
80222|         #if DO_ALL_SEARCH
80223|
    | File_PrintOneLiner("SwapFile",DeviceObject,Irp);
80224|         #endif
80225|         goto SendOrig;
80226|     }
80227|
80228| #if 0
80229| // rob - we now write to our files direct during the
    | mount/dismount stage
80230|     // FIXFIXFIX This is a temporary flag to
    | prevent caching during rebuild - to be replaced
80231|     // with a holding pen to save the writes till
    | after rebuild is through.
80232|     // dont psm writes until the rebuild code has
    | completely initialized the system.
80233|     if ( DevExt->InLoadUnload ) {
80234|
    | File_PrintOneLiner("InLoadUnload",DeviceObject,Irp);
80235|         goto SendOrig;
80236|     }
80237| #endif
80238|
80239|     // if device has already been shutdown, then
    | dont handle it.
80240|     if ( DevExt->DeviceShutDown ) {
80241|
    | File_PrintOneLiner("ShutDown",DeviceObject,Irp);
80242|         goto SendOrig;
80243|     }
80244|
80245|     // make sure to keep in sync with
    | SFliter:Sfwrite
80246|     if(DevExt->Cache.CacheFullAction) {
80247|         ULONG CacheThreshold = (ULONG)((((unsigned
    | __int64)DevExt->Cache.PSManBitMapSize *
    | DevExt->Cache.CacheFullActionPercent) / 100);
80248|         if (

```

```

    | DevExt->Cache.CurrentCacheFileSize>CacheThreshold ) {
80249|
    | File_PrintOneLiner("CacheFull",DeviceObject,Irp);
80250|         switch(DevExt->Cache.CacheFullAction) {
80251|             case CACHE_ACTION_DENY_WRITES :
80252|
            | if(AreThereAlwaysKeepSnapShots()) {
80253|
            | Debug(DEBUG_WRITE,("PSManWriteDevice: Reporting
            | STATUS_DISK_FULL because of always-keep snapshots.
            | DevExt=%08x\n",DevExt));
80254|                 Irp->IoStatus.Status =
            | Status = STATUS_DISK_FULL;
80255|                 Irp->IoStatus.Information =
            | 0;
80256|
            | IoCompleteRequest(Irp,IO_NO_INCREMENT);
80257|                 InterlockedDecrement(
            | (PLONG)&OutstandingRequests );
80258|
            | ReleaseGlobalDeviceForRead();
80259|                 return Status;
80260|             }
80261|             break;
80262|             case CACHE_ACTION_BSOD :
80263|
            | PSManBugCheck(SB_BUG_WRITE_FILE,SB_CACHE_FULL,DevExt->Ca
            | che.CurrentCacheFileSize,CacheThreshold,DevExt->Cache.Ca
            | cheFullActionPercent);
80264|             break;
80265|             case
            | CACHE_ACTION_DELETE_ALWAYS_KEEPS:
80266|                 break;
80267|             default:
80268|                 break;
80269|             }
80270|         }
80271|     }
80272|
80273|
80274|     //Debug(DEBUG_WRITE,("Sector %d being
            | saved\n",Sector));
80275|
80276|     WriteRequest = (tWriteRequest *)
            | MemAllocatePoolWithTag(NonPagedPool,
80277|
            | sizeof(tWriteRequest),
80278|
            | WRITEREQUESTTAG);
80279|     if ( !WriteRequest ) {

```

```

80280|         Debug(DEBUG_WRITE | DEBUG_ERROR,("Error!
      | Out of memory allocating buffer for write\n"));
80281|
      | FailRequest(NULL,STATUS_INSUFFICIENT_RESOURCES);
80282|
80283|         //DbgBreakPoint();
80284|
80285|         SendOrig:
80286|         //Debug(DEBUG_WRITE | DEBUG_INFO,("Calling
      | original driver with original irp %p\n",Irp));
80287|
80288|         pmAcquireSpinLock (
      | &DevExt->StatisticsSpinLock, &oldIrql );
80289|         // update the logical partition
80290|         DevExt->SectorsWritten +=
      | (currentIrpStack->Parameters.Write.Length /
      | DevExt->BytesPerSector);
80291|         DevExt->NumberOfWriteRequests++;
80292|
80293|         // update the physical drive
80294|         //PhyExt->SectorsWritten +=
      | (currentIrpStack->Parameters.Write.Length /
      | PhyExt->BytesPerSector);
80295|         //PhyExt->NumberOfWriteRequests++;
80296|         pmReleaseSpinLock (
      | &DevExt->StatisticsSpinLock, oldIrql );
80297|
80298|         SendOrigNoAudit:
80299|         InterlockedDecrement(
      | (PLONG)&OutstandingRequests );
80300|         ReleaseGlobalDeviceForRead();
80301|         try_return(Status = PSMANPassThru(
      | DeviceObject, Irp ));
80302|     }
80303|
80304|
80305|     /*
80306|         Debug(DEBUG_INFO,("Write:
      | PsGetCurrentProcess=%08x "
80307|         "PsGetCurrentThread=%08x "
80308|         "IoGetCurrentProcess=%08x
      | "
80309|         "KeGetCurrentThread=%08x "
80310|         "User Thread=%08x\n",
80311|         PsGetCurrentProcess(),
80312|         PsGetCurrentThread(),
80313|         IoGetCurrentProcess(),
80314|         KeGetCurrentThread(),
80315|         Irp->Tail.Overlay.Thread
80316|         ));

```

```

80317|    */
80318|
80319|    FillInWriteRequest( WriteRequest, DeviceObject,
    | Irp, DevExt->BytesPerSector);
80320|
80321|    if ( !IsBeingProcessed(WriteRequest) ) {
80322|        pDictionary Dict;
80323|
80324|        // note:
80325|        //      The snapshot resource may
    | already be acquired
80326|        //      because a thread did
    | SbWriteAndWait, but before
80327|        //      the write is sent to me, ntfs
    | decides to checkpoint
80328|        //      the volume.
80329|        //      We acquire this resource so the
    | dictionary is not
80330|        //      deleted while we are accessing it.
80331|
80332| // rob - 11-6-2000 we cant acquire the snapshot
    | resource after the global resource
80333| // as a deadlock will occur when waiting for quiesence.
80334| //      GetSnapShotForRead();
80335| //      __try {
80336|
    | PersistentDictionary::GetDictionaryForVolume(DeviceObjec
    | t,Dict);
80337|    if ( Dict ) {
80338|        ULARGE_INTEGER SectorHuge;
80339|        ULONG CountDid=0;
80340|
80341|        SectorHuge.QuadPart =
    | WriteRequest->RealSector.QuadPart;
80342|
80343|        if (
    | PersistentDictionary::DoFreeSpaceChecks() ) {
80344|            if (
    | (((pPersistentDictionary)Dict)->NeedsCaching(SectorHuge,
    | WriteRequest->RealCount)) ) {
80345| #if 0
80346|            // rob - 4-7-2001 - commented
    | out, as we now keep our rbtree stuff in virtual
80347|            // memory, and we cant access
    | that from an arbitrary thread context.
80348|            // This was just trying to
    | prevent a read from disk and 2 thread switches.
80349|            // SaveOriginal* does the same
    | thing, and will discard the data.
80350|            // I dont think this actually

```

```

| would find anything anyway
80351|         CheckForCache:
80352|         // can this test go now we have
| the new Granule bit map logic??    FIXFIXFIX think
| needed !!  see if "Already cached" messages
| disappear!!!!
80353|         Status =
| ((pPersistentDictionary)Dict)->searchMultiple(
80354|
| DevExt,
80355|
| SectorHuge,
80356|
| WriteRequest->RealCount,
80357|
| CountDid,
80358|
| NULL,
80359|
| SectorHuge, // this doesnt matter since we are passing
| in a NULL buffer
80360|
| NULL,
80361|
| 0);
80362|
80363|         if ( (Status==STATUS_SUCCESS)
| && (CountDid == WriteRequest->RealCount) ) {
80364|             #if DO_ALL_SEARCH
80365|
| File_PrintOneLiner("AlreadyCached",DeviceObject,Irp);
80366|             #endif /*DO_ALL_SEARCH*/
80367|             FREE_POINTER(WriteRequest);
80368|             goto SendOrig;
80369|         }
80370| #endif
80371|         } else {
80372|             // set that this data needs to
| be psmed on the next write
80373|             // no no no!!
| ((pPersistentDictionary)Dict)->SetFreeSpaceStatus(Sector
| Huge,WriteRequest->RealCount,TRUE);
80374|             #if DO_ALL_SEARCH
80375|
| File_PrintOneLiner("FreeSpace &/or
| cached",DeviceObject,Irp);
80376|             #endif /*DO_ALL_SEARCH*/
80377|             FREE_POINTER(WriteRequest);
80378|             goto SendOrig;
80379|         }

```

```

80380|         } else { // DoFreeSpaceChecks
80381| #if 0
80382|             goto CheckForCache;
80383| #endif
80384|         }
80385|     }
80386| //     } __finally {
80387| //         ReleaseSnapShotForRead();
80388| //     }
80389|     } else {
80390|     }
80391|
80392| #ifdef DEBUG
80393| //     File_Printf("Saving",DeviceObject,Irp);
80394|     #if DO_ALL_SEARCH
80395|
80396|         | File_PrintOneLiner("Saving",DeviceObject,Irp);
80397|     #endif /*DO_ALL_SEARCH*/
80398| #endif
80399|     // at least one or more need to be psmmed
80400|     // or we cant tell because it is being
80401|     | processed
80402|     // by another thread
80403|
80404|     //Debug(DEBUG_WRITE,("WriteDevice:
80405|     | WriteRequest=%08x, DeviceObject=%08x,
80406|     | Irp=%08x\n",WriteRequest,DeviceObject,Irp));
80407|     pmAcquireSpinLock ( &WriteSpinLock, &oldIrql );
80408|     // add to queue...
80409|     InsertTailList (
80410|         | &WriteQueue,&(WriteRequest->ListEntry));
80411|     WriteQueueDepth++;
80412|     if ( WriteQueueDepth>MaxWriteQueueDepth ) {
80413|         MaxWriteQueueDepth = WriteQueueDepth;
80414|     }
80415|     pmReleaseSpinLock( &WriteSpinLock, oldIrql );
80416|     // mark the current request as pending...
80417|     IoMarkIrpPending(Irp);
80418|
80419|     //Debug(DEBUG_WRITE,("Semaphore Count =
80420|     | %d\n",WriteSemaphore.Header.SignalState));
80421|     // have a thread work on it.
80422|     pmSignalSemaphore( &WriteSemaphore);
80423|     Status = STATUS_PENDING;

```

```

80424|     ReleaseGlobalDeviceForRead();
80425|     try_exit: NOTHING;
80426| } __except
    | (ExceptionFilter(GetExceptionInformation())) {
80427|     Irp->IoStatus.Status = Status =
    | GetExceptionCode();
80428|     Debug(DEBUG_WRITE,("PSMan: Write: Exception
    | %08x\n",Status));
80429|     Irp->IoStatus.Information = 0;
80430|     IoCompleteRequest(Irp,IO_NO_INCREMENT);
80431|     ReleaseGlobalDeviceForRead();
80432|     FailRequest(NULL,Status);
80433| }
80434|
80435| return Status;
80436| // return STATUS_MORE_PROCESSING_REQUIRED;
80437|
80438| } // end PSManWriteDevice()
80439|
80440| STATIC ULONG IKnowAboutTheNtfsStructures = 1;
80441|
80442| /*-----
    | -----*/
80443| STATIC NTSTATUS TdWaitForWriteWork()
80444| {
80445|     PVOID ObjectTable[2] = { &VDiskExitingEvent,
    | &WriteVDiskSemaphore};
80446|
80447|     ASSERT(KeGetCurrentIrql() < DISPATCH_LEVEL);
80448|     return
    | pmWaitForMultipleObjects(ObjectTable,2,NULL);
80449| }
80450|
80451| /*-----
    | -----*/
80452| STATIC NTSTATUS WritePreProcessSpecialSectors (
80453|     | PDEVICE_OBJECT DeviceObject,
80454|     | ULARGE_INTEGER LogSector,
80455|     | ULONG
    | Count,
80456|     | char
    | *Buffer )
80457| {
80458|     NOT_REFERENCED(DeviceObject);
80459|     NOT_REFERENCED(LogSector);
80460|     NOT_REFERENCED(Count);
80461|     NOT_REFERENCED(Buffer);
80462|     return STATUS_SUCCESS;

```

```

80463| }
80464|
80465| /*-----
      | -----*/
80466| STATIC NTSTATUS WritePostProcessSpecialSectors (
80467|     | PDEVICE_OBJECT DeviceObject,
80468|     | ULARGE_INTEGER LogSector,
80469|     | ULONG
      | Count,
80470|     | char
      | *Buffer )
80471| {
80472|     NOT_REFERENCED(DeviceObject);
80473|     NOT_REFERENCED(LogSector);
80474|     NOT_REFERENCED(Count);
80475|     NOT_REFERENCED(Buffer);
80476|     return STATUS_SUCCESS;
80477| }
80478|
80479| /*-----
      | -----*/
80480|
80481| STATIC long CacheThresholdReached_VDisk_NumPending = 0;
80482|
80483| void CacheThresholdReached_VDisk_Thread ( PVOID Context
      | )
80484| {
80485|     long NumPending = InterlockedDecrement
      | (&CacheThresholdReached_VDisk_NumPending);
80486|     ASSERT(NumPending >= 0);
80487|     PDEVICE_OBJECT Volume = (PDEVICE_OBJECT) Context;
80488|
      | Debug(DEBUG_WRITE,("CacheThresholdReached_VDisk_Thread:
      | Volume=%08x, NumPending=%08x\n",Volume,NumPending));
80489|     CacheThresholdReached (Volume);
80490| }
80491|
80492| /*-----
      | -----*/
80493|
80494| void CacheThresholdReached_VDisk (PDEVICE_OBJECT
      | Volume)
80495| {
80496|     long NumPending =
      | InterlockedIncrement(&CacheThresholdReached_VDisk_NumPen
      | ding);
80497|     bool ThreadWasStarted = false;
80498|     ASSERT(NumPending > 0);

```



```

80499|  ASSERT(NumPending <= 2);
80500|  Debug(DEBUG_WRITE,("CacheThresholdReached_VDisk;
      | Volume=%08x, NumPending=%08x\n",Volume,NumPending));
80501|  if ( NumPending <= 1 ) {
80502|      HANDLE ThreadHandle = INVALID_HANDLE_VALUE;
80503|
80504|      NTSTATUS status = pmStartThread (
80505|          (PKSTART_ROUTINE)
      | CacheThresholdReached_VDisk_Thread,
80506|          Volume,
80507|          &ThreadHandle );
80508|
80509|      if ( NT_SUCCESS(status) ) {
80510|          ZwClose (ThreadHandle);
80511|          ThreadHandle = INVALID_HANDLE_VALUE;
80512|          ThreadWasStarted = true;
80513|      } else {
80514|          ASSERT(FALSE);
80515|      }
80516|  }
80517|
80518|  if ( !ThreadWasStarted ) {
80519|      NumPending =
      | InterlockedDecrement(&CacheThresholdReached_VDisk_NumPen
      | ding);
80520|      ASSERT(NumPending >= 0);
80521|  }
80522| }
80523|
80524| /*-----
      | -----*/
80525| void WriteVDiskThread ( PVOID Context )
80526| {
80527|     ULONG Exiting=0;
80528|     NTSTATUS Status=0;
80529|     ULONG ThresholdReached = FALSE;
80530|
80531|     PAGED_CODE();
80532|
80533|     pmAcquireMutex ( &VDiskThreadMutex, NULL);
80534|     VDiskNumberOfThreads++;
80535|     pmReleaseMutex ( &VDiskThreadMutex);
80536|
80537|     pmWaitForSingleObject( &Thread0Initd, NULL);
80538|
80539|     // say we are initd
80540|     pmSetEvent( (PKEVENT)Context );
80541|
80542|     RestartThreadFromError:
80543|

```

```

80544|  __try {
80545|      while ( !Exiting ) {
80546|          Status = TdWaitForWriteWork();
80547|
80548|          if ( Status == STATUS_WAIT_1 ) {
80549|              PLIST_ENTRY ListEntry=NULL;
80550|              GetAnother:
80551|              ListEntry = ExInterlockedRemoveHeadList
80552|              | (
80553|              | &WriteVDiskQueue,    // List Head
80554|              | &WriteVDiskSpinLock // Lock
80555|              | );
80556|              PVDISK_EXTENSION DevExt=NULL;
80557|              if ( (ListEntry) &&
80558|              | (ListEntry!=&WriteVDiskQueue) ) {
80559|                  tWriteRequest
80560|                  | *WriteRequest=NULL;
80561|                  PIO_STACK_LOCATION
80562|                  | currentIrpStack=NULL;
80563|                  CHAR          IoIncrement =
80564|                  | IO_NO_INCREMENT;
80565|                  char          *Buffer=NULL;
80566|                  KIRQL          oldIrql;
80567|
80568|                  /*lint -save -e413 */
80569|                  WriteRequest = CONTAINING_RECORD(
80570|                  | ListEntry, tWriteRequest, ListEntry );
80571|                  /*lint -restore */
80572|                  DevExt =
80573|                  | (PVDISK_EXTENSION)GetDeviceExtension(WriteRequest->Devic
80574|                  | eObject);
80575|                  ASSERT(DevExt->ObjectType==OBJECT_VIRTUALDISK);
80576|
80577|                  if (
80578|                  | IsBeingProcessedEx(WriteRequest) ) {
80579|                      BOOLEAN OnlyOne=FALSE;
80580|
80581|                      pmAcquireSpinLock (
80582|                      | &WriteVDiskSpinLock, &oldIrql );
80583|                      InsertTailList
80584|                      | (&WriteVDiskQueue,&WriteRequest->ListEntry);
80585|
80586|                      // if only one on list

```

```

80579|             if ( WriteVDiskQueue.Flink ==
| &WriteRequest->ListEntry ) {
80580|                 OnlyOne=TRUE;
80581|             }
80582|             pmReleaseSpinLock(
| &WriteVDiskSpinLock, oldIrql );
80583|             if ( OnlyOne ) {
80584|                 LARGE_INTEGER TimeToWait =
| {0};
80585|                 TimeToWait.QuadPart =
| RELATIVE(MILLISECONDS(1));
80586|                 KeDelayExecutionThread(
| (KPROCESSOR_MODE)KernelMode, FALSE, &TimeToWait );
80587|             }
80588|             goto GetAnother;
80589|         }
80590|
80591|
80592|             // make sure PSM is enabled for
| this device.. otherwise someone is accessing us
80593|             // without our approval...
80594| //
| ASSERT(((PFILTERED_EXTENSION)(GetDeviceExtension(DevExt-
| >PSMDevice)))->PSMed);
80595|
80596|             currentIrpStack =
| IoGetCurrentIrpStackLocation(WriteRequest->Irp);
80597|
80598|
| WriteRequest->Irp->IoStatus.Information = 0;
80599|
80600|             // acquire resource first or a
| deadlock will occur
80601|             GetSnapShotForRead();
80602|             __try {
80603|                 // since we are waiting for a
| read, it may have been deleted
80604|                 // make sure
80605|                 if ( DevExt->SnapShot ) {
80606|
| //Debug(DEBUG_WRITE,("VDisk: Write: Acquiring VDisk
| resource\n"));
80607|                 AcquireVDiskResource();
80608|                 __try {
80609|                     // protect this area so
| we dont bring down NT
80610|                     __try {
80611|
80612|                         // make sure the
| device didnt disappear while

```

```

80613|                // waiting for
| mutex
80614|                if (
| (DevExt->PartitionActive)/* &&
| (!GlobalData->ShutDownCalled)*/ ) {
80615|
80616|                // if a write
80617|                if (
| currentIrpStack->MajorFunction == IRP_MJ_WRITE ) {
80618| #if DO_ALL_IO
80619|
| PUNICODE_STRING FileName;
80620|
| //Debug(DEBUG_WRITE,("VDisk: Write: Irp %p F=%08x-FO %p
| F=%08x-Sf=%08x
| \n",Irp,Irp->Flags,Irp->Tail.Overlay.OriginalFileObject,
| currentIrpStack->Flags));
80621|
| FileName=File_GetFullFileName(WriteRequest->Irp->Tail.Ov
| erlay.OriginalFileObject);
80622|
| Debug(DEBUG_WRITE,("VDisk: Write : %p-%p Log (%p)=%l64x
| for %03x
| '%wZ'\n",WriteRequest->Irp,WriteRequest->Irp->Tail.Overl
| ay.OriginalFileObject,WriteRequest->DeviceObject,WriteRe
| quest->RealSector,WriteRequest->RealCount,FileName));
80623|                if (
| FileName ) {
80624|
| FREE_POINTER(FileName);
80625|                }
80626| #endif
80627|                if (
| WriteRequest->Irp->MdlAddress != NULL ) {
80628| #if _WIN32_WINNT >=0x0500
80629|                Buffer
| = (char *)MmGetSystemAddressForMdlSafe(
| WriteRequest->Irp->MdlAddress, NormalPagePriority );
80630| #else
80631|                Buffer
| = (char *)MmGetSystemAddressForMdl(
| WriteRequest->Irp->MdlAddress );
80632| #endif
80633|
| WritePreProcessSpecialSectors(
80634|
| WriteRequest->DeviceObject,
80635|
| WriteRequest->RealSector,
80636|

```

```

    | WriteRequest->RealCount,
80637|
    | Buffer );
80638|
80639|
    | ASSERT(WriteRequest->RoundedCountInBytes.HighPart==0);
80640|
80641|
80642| /*
80643|     Virtual Write, not in Virtual cache, not in cache -
    | Read old data, add to cache
80644|     Virtual Write, not in Virtual cache, in cache -
80645|     Virtual Write, in Virtual cache, in cache -
80646|     Virtual Write, in Virtual cache, in cache -
80647| */
80648|
80649|                                     PCHAR
    | NewBuffer =
    | (PCHAR)MemAllocatePoolWithTag(PagedPool,WriteRequest->Ro
    | undedCountInBytes.LowPart,PSM_VDISK_BUFFER_TAG);
80650|                                     if (
    | NewBuffer ) {
80651|
    | PMDL NewMdl = IoAllocateMdl(
    | NewBuffer,WriteRequest->RoundedCountInBytes.LowPart,FALS
    | E,FALSE,NULL);
80652|                                     if
    | ( NewMdl ) {
80653|
    | /*lint -save -e149 */
80654|
    | __try {
80655|
    | //MmBuildMdlForNonPagedPool(Mdl);
80656|
    | MmProbeAndLockPages(NewMdl,
    | (KPROCESSOR_MODE)KernelMode,IoModifyAccess);
80657|
    | } __except( ExceptionFilter(GetExceptionInformation())
    | ) {
80658|
    | Debug(DEBUG_THREAD,("Exception while locking Mdl %p for
    | %p\n",NewMdl,NewBuffer));
80659|
    | Status = GetExceptionCode();
80660|
    | }
80661|
    | /*lint -restore */
80662|

```

```

80663|
    | if ( NT_SUCCESS(Status) ) {
80664|
        | LARGE_INTEGER UL;
80665|
        | UL.QuadPart = WriteRequest->RoundedSector.QuadPart *
        | DevExt->BPS;
80666|
80667|
        | // read in whole granules
80668|
        | Status = Sblo_ReadDeviceMdl(
80669|
        | DevExt->PSMDevice, // PDEVICE_OBJECT DeviceObject,
80670|
        | &UL, // PLARGE_INTEGER ByteOffset,
80671|
        | WriteRequest->RoundedCountInBytes.LowPart, // ULONG
        | ByteCount,
80672|
        | WriteRequest->Irp, // PIRP          OriginalIrp,
80673|
        | NewMdl // PMDL          Mdl
80674|
        | );
80675|
80676|
        | if ( NT_SUCCESS(Status) ) {
80677|
            | ULARGE_INTEGER BS;
80678|
            | ULARGE_INTEGER UL;
80679|
            | ULONG CountDid=0;
80680|
            | BS.QuadPart =
            | WriteRequest->RoundedCountInBytes.QuadPart;
80681|
80682|
            | UL.QuadPart = WriteRequest->RoundedSector.QuadPart;
80683|
            | Status = DevExt->SnapShot->Dictionary->searchMultiple(
80684|
            | (PFILTERED_EXTENSION)GetDeviceExtension(DevExt->PSMDevic
            | e),
80685|
            | UL,
80686|
            | WriteRequest->RoundedCount,
80687|

```

```

    | CountDid,
80688|
    | NULL,
80689|
    | BS,
80690|
    | NewBuffer,
80691|
    | DICT_FLAG_VIRTUAL_IO);
80692|
80693|
    | if ( Status == STATUS_NOT_FOUND ) {
80694|
    | Status = STATUS_SUCCESS; // Not really a problem.
80695|
    | }
80696|
80697|
    | if ( !NT_SUCCESS(Status) ) {
80698|
    | Debug(DEBUG_WRITE,("Vdisk Write: error %08x getting old
    | data\n",Status));
80699|
    | //ASSERT(FALSE);
80700|
    | }
80701|
80702|
    | RtlMoveMemory(
80703|
    | &NewBuffer[(WriteRequest->RealSector.QuadPart-WriteReque
    | st->RoundedSector.QuadPart)*DevExt->BPS],
80704|
    | Buffer,
80705|
    | WriteRequest->RealCount*DevExt->BPS);
80706|
80707|
    | if ( NT_SUCCESS(Status) ) {
80708|
    | // Check for using too much cache...
80709|
    | if (
    | pPersistentDictionary(DevExt->SnapShot->Dictionary)->IsC
    | acheWarningThresholdReached() ) {
80710|
    | ThresholdReached = TRUE;
80711|
    | }
80712|

```

```

    | }
80713|
80714|
    | if ( NT_SUCCESS(Status) ) {
80715|
    | Status = AddSectorsToMemoryCache(
80716|
    | DevExt,
80717|
    | WriteRequest->RoundedSector,
80718|
    | WriteRequest->RoundedCount,
80719|
    | NewBuffer);
80720|
    | if ( !NT_SUCCESS(Status) ) {
80721|
    | Debug(DEBUG_WRITE,("Vdisk Write: Add to cache failed
    | error %08x\n",Status));
80722|
80723|
    | // always assume success
80724|
    | Status = STATUS_SUCCESS;
80725|
    | }
80726|
    | }
80727|
80728|
    | //Debug(DEBUG_WRITE,("VDisk: Write: Log (%p)=%d for %d
    | Done\n",DeviceObject,LogSector,Count));
80729|
80730|
    | if ( NT_SUCCESS(Status) ) {
80731|
    | IoIncrement = IO_DISK_INCREMENT;
80732|
    | WriteRequest->Irp->IoStatus.Information =
    | WriteRequest->ByteLength;
80733|
    | WritePostProcessSpecialSectors (
80734|
    | WriteRequest->DeviceObject,
80735|
    | WriteRequest->RealSector,
80736|
    | WriteRequest->RealCount,
80737|
    | Buffer );

```



```

80738|
80739|
    | // Update stats about volume
80740|
    | // dont need spin lock as only 1 write can occur at any
    | time
80741|
    | DevExt->SectorsWritten+=WriteRequest->RealCount;
80742|
    | DevExt->NumberOfWriteRequests++;
80743|
80744|
    | }
80745|
    | } else { // success of ReadDeviceMdl
80746|
    | Debug(DEBUG_WRITE,("VDisk: Write: Error %08x reading
    | via mdl\n",Status));
80747|
    | }
80748|
    | MmUnlockPages(NewMdl);
80749|
    | } else { // status of mdl
80750|
    | Debug(DEBUG_WRITE,("VDisk: Write: Unable to lock
    | mdl\n"));
80751|
    | }
80752|
    | IoFreeMdl(NewMdl);
80753|                                     }
    | else { // alloc mdl
80754|
    | Debug(DEBUG_WRITE,("VDisk: Write: Unable to alloc new
    | mdl\n"));
80755|
    | Status = STATUS_INSUFFICIENT_RESOURCES;
80756|                                     }
80757|
    | MemFreePool(NewBuffer);
80758|                                     } else
    | { // buffer
80759|
    | Debug(DEBUG_WRITE,("VDisk: Write: Out of memory for
    | buffer\n"));
80760|
    | Status = STATUS_INSUFFICIENT_RESOURCES;
80761|                                     }
80762|                                     } else {

```

```

80763|
| Debug(DEBUG_WRITE,("VDisk: Write: Invalid MDL\n"));
80764|                                     Status
| = STATUS_INVALID_USER_BUFFER;
80765|                                     }
80766|                                     } else {
80767|
| Debug(DEBUG_WRITE,("VDisk: Verify: Log (%p)=%l64d for
| %d\n",WriteRequest->DeviceObject,WriteRequest->RealSecto
| r.QuadPart, WriteRequest->RealCount));
80768|                                     Status =
| STATUS_SUCCESS;
80769|
| WriteRequest->Irp->IoStatus.Information =
| WriteRequest->ByteLength;
80770|                                     }
80771|                                     } else {
80772|                                     if (
| GlobalData->ShutDownCalled ) {
80773|
| Debug(DEBUG_WRITE,("VDisk: Write: Shutdown already
| called\n"));
80774|
| WriteRequest->Irp->IoStatus.Information =
| WriteRequest->ByteLength;
80775|                                     Status = 0;
80776|                                     } else {
80777|
| Debug(DEBUG_WRITE,("VDisk: Write: Device disappeared or
| cache file not open\n"));
80778|
| WriteRequest->Irp->IoStatus.Information =
| INFORMATION_FOR_NOT_ACTIVE(Irp);
80779|                                     Status =
| VOLUME_NOT_ACTIVE_ERROR_CODE;
80780|                                     }
80781|                                     }
80782|
80783|                                     // if media error,
| say so...
80784|                                     switch ( Status ) {
80785|                                     case
| STATUS_MEDIA_CHANGED :
80786|                                     case
| STATUS_NO_MEDIA_IN_DEVICE : {
80787|
| InterlockedIncrement((PLONG)&DevExt->DiskChangeCount);
80788|
| DevExt->DriveNotReady = TRUE;
80789|

```

```

80790|
| Debug(DEBUG_WRITE,("VDisk: Write: Media may have
| changed\n"));
80791|                                     if (
| DevExt->DeviceObject->Vpb->Flags & VPB_MOUNTED ) {
80792|
| Debug(DEBUG_WRITE,("VDisk: Write: Informing File system
| Me=%08x, DO=%08x,
| RO=%08x\n",DevExt->DeviceObject,DevExt->DeviceObject->Vp
| b->DeviceObject,DevExt->DeviceObject->Vpb->RealDevice));
80793|
| DevExt->DeviceObject->Flags |= DO_VERIFY_VOLUME;
80794|
| Status = STATUS_VERIFY_REQUIRED;
80795|                                     } else
| {
80796|                                     //
| we may need to do this.
80797|                                     //
| Status = STATUS_IO_DEVICE_ERROR;
80798|                                     }
80799|                                     break;
80800|                                     }
80801|                                     default:
80802|                                     break;
80803|                                     }
80804|                                     }
| __except(ExceptionFilter(GetExceptionInformation())) {
80805|                                     Status =
| GetExceptionCode();
80806|
| Debug(DEBUG_WRITE,("VDisk: Write: Exception
| %08x\n",Status));
80807|                                     }
80808|                                     } __finally {
80809|                                     ReleaseVDiskResource();
80810|
| WriteRequest->Irp->IoStatus.Status = Status;
80811|                                     // TESTTEST keep "lost
| delayed write" messages from popping up
80812| #if 1
80813|                                     if (
| IoIsErrorUserInduced(Status) ) {
80814|
| Debug(DEBUG_WRITE,("Setting hard error for error
| %08x\n",Status));
80815|
| IoSetHardErrorOrVerifyDevice(WriteRequest->Irp,DevExt->D
| eviceObject);
80816|                                     }

```

```

80817| #endif
80818|             if ( Status!=0 ) {
80819|
80820| | Debug(DEBUG_WRITE,("Vdisk: Write: Device %08x Irp %08x
80821| | Error
80822| | %08x\n",DevExt->DeviceObject,WriteRequest->Irp,Status));
80823|
80824|             }
80825|             IoCompleteRequest
80826| | (WriteRequest->Irp, IoIncrement) ;
80827|
80828|             }
80829|             } else {
80830|
80831| | Debug(DEBUG_WRITE,("VdiskWrite: SnapShot Deleted while
80832| | waiting or not psmmed\n"));
80833|
80834| | WriteRequest->Irp->IoStatus.Status =
80835| | VOLUME_NOT_ACTIVE_ERROR_CODE;
80836|             IoCompleteRequest
80837| | (WriteRequest->Irp, IoIncrement) ;
80838|
80839|             }
80840|             } __finally {
80841|             ReleaseSnapShotForRead();
80842|             }
80843|
80844| | pmAcquireSpinLock(&WriteSpinLock,&oldIrql);
80845|
80846| | RemoveEntryList(&(WriteRequest->ProcessingEntry));
80847|
80848| | pmReleaseSpinLock(&WriteSpinLock,oldIrql);
80849|             FREE_POINTER(WriteRequest);
80850|             } else {
80851|             Debug(DEBUG_WRITE,("Vdisk: Write:
80852| | Error ListEntry is NULL\n"));
80853|             }
80854|
80855|             if ( ThresholdReached ) {
80856|             ThresholdReached = FALSE;
80857|             ASSERT(DevExt != NULL);
80858|             if ( DevExt != NULL ) {
80859|             ASSERT(DevExt->SnapShot !=
80860| | NULL);
80861|             if ( DevExt->SnapShot != NULL )
80862|             | {
80863|             | ASSERT(DevExt->SnapShot->DeviceObject != NULL);
80864|             | if (
80865|             | | DevExt->SnapShot->DeviceObject != NULL ) {
80866|             |
80867|             | CacheThresholdReached_VDisk(DevExt->SnapShot->DeviceObje

```

```

    | ct);
80849|         }
80850|     }
80851| }
80852| }
80853| } else {
80854|     Exiting = 1;
80855| }
80856| }
80857| }
    | __except(ExceptionFilter(GetExceptionInformation())) {
80858|     Debug(DEBUG_WRITE,("VDisk: Write: Exception
    | %08x in thread\n",GetExceptionCode()));
80859| }
80860|
80861| // cant goto from within exception handler...
80862| if ( !Exiting ) {
80863|     goto RestartThreadFromError;
80864| }
80865|
80866| //ExitThread:
80867|
80868|     Debug(DEBUG_WRITE,("VdiskWrite: Exiting\n"));
80869|
80870|     // free cache file handle
80871|     AcquireVDiskResource();
80872|     __try {
80873|
80874|         // free any writes to the volume
80875|         while ( !IsListEmpty(&WriteVDiskQueue) ) {
80876|             PLIST_ENTRY ListEntry;
80877|             tWriteRequest *WriteRequest;
80878|             PIRP Irp;
80879|             KIRQL oldIrql;
80880|
80881|             Debug(DEBUG_READ,("VdiskWrite: Cleaning up
    | write on vdisk queue\n"));
80882|             ListEntry = ExInterlockedRemoveHeadList (
80883|
    | &WriteVDiskQueue,    // List Head
80884|
    | &WriteVDiskSpinLock // Lock
80885|             );
80886|             /*lint -save -e413 */
80887|             WriteRequest = CONTAINING_RECORD(
    | ListEntry, tWriteRequest, ListEntry );
80888|             /*lint -restore */
80889|
80890|             Irp      = WriteRequest->Irp;
80891|             pmAcquireSpinLock(&WriteSpinLock,&oldIrql);

```

```

80892|
| RemoveEntryList(&(WriteRequest->ProcessingEntry));
80893|         pmReleaseSpinLock(&WriteSpinLock,oldIrp);
80894|         FREE_POINTER(WriteRequest);
80895|
80896|         Irp->IoStatus.Information =
| INFORMATION_FOR_NOT_ACTIVE(Irp);
80897|         Irp->IoStatus.Status =
| VOLUME_NOT_ACTIVE_ERROR_CODE;
80898|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
80899|     }
80900| } __finally {
80901|     ReleaseVDiskResource();
80902| }
80903|
80904| pmAcquireMutex ( &VDiskThreadMutex, NULL);
80905| VDiskNumberOfThreads--;
80906| pmReleaseMutex ( &VDiskThreadMutex);
80907|
80908| Debug(DEBUG_WRITE,("VdiskWrite: Exited\n"));
80909| PsTerminateSystemThread( 0 );
80910| }
80911|
80912| STATIC PFILE_OBJECT
| NtFsMetaDataFileObjects[MAX_NTFS_ENTRY_NUM]={0};
80913|
80914|
80915| /*-----
| -----*/
80916| STATIC NTSTATUS PSMANWriteVDisk(
80917|         IN PDEVICE_OBJECT
| DeviceObject,
80918|         IN PIRP Irp
80919| )
80920| {
80921|     PIO_STACK_LOCATION currentIrpStack =
| IoGetCurrentIrpStackLocation(Irp);
80922|     NTSTATUS Status=STATUS_PENDING;
80923|     tWriteRequest *WriteRequest=NULL;
80924|     PVDISK_EXTENSION
| DevExt=(PVDISK_EXTENSION)GetDeviceExtension(DeviceObject
| );
80925|
80926|     PAGED_CODE();
80927|
80928| //     Debug(DEBUG_WRITE | DEBUG_PROCCALL,
| ("PSMANWriteVDisk Called\n"));
80929|
80930|     ASSERT(DevExt->ObjectType==OBJECT_VIRTUALDISK);
80931|

```

```

80932| // TESTTEST
80933| #define _COMPLETE_WRITES_UNCONDITIONALLY_ 0
80934|
80935| #if _COMPLETE_WRITES_UNCONDITIONALLY_
80936|     Status = STATUS_SUCCESS;
80937|     Irp->IoStatus.Information =
        | currentIrpStack->Parameters.Write.Length;
80938|     IoCompleteRequest(Irp,IO_DISK_INCREMENT);
80939|     return Status;
80940| #endif
80941|
80942|
80943| #if DO_ERROR_WRITES
80944|     if ( CheckMediaLoaded(DeviceObject, Irp
        | )!=STATUS_SUCCESS ) {
80945|         Debug(DEBUG_WRITE,("VDisk: Write: Media not
        | loaded\n"));
80946|         Status = Irp->IoStatus.Status;
80947|         IoCompleteRequest(Irp,IO_NO_INCREMENT);
80948|         return Status;
80949|     }
80950|
80951| #else
80952|     // trying to minimize "lost delayed write" errors
80953|     if ( (!DevExt->PartitionActive) ||
        | (DevExt->DriveNotReady) ) {
80954|         // when a error occurred (out of cache file for
        | example...)
80955|         Debug(DEBUG_WRITE,("VDisk: Write: drive not
        | ready\n"));
80956|         Irp->IoStatus.Information =
        | INFORMATION_FOR_NOT_ACTIVE(Irp);
80957|         Irp->IoStatus.Status = Status =
        | VOLUME_NOT_ACTIVE_ERROR_CODE;
80958|         IoCompleteRequest(Irp,IO_DISK_INCREMENT);
80959|         return Status;
80960|     }
80961| #endif
80962|
80963| /* TESTTEST
80964|     // trying to minimize "lost delayed write" errors
80965|     if(!DevExt->PSMed) {
80966|         // when a error occurred (out of cache file for
        | example...)
80967|         Debug(DEBUG_WRITE,("VDisk: Write: drive not
        | psmed\n"));
80968|         Irp->IoStatus.Information =
        | IoGetCurrentIrpStackLocation(Irp)->Parameters.Write.Leng
        | th;
80969|         Irp->IoStatus.Status = Status = STATUS_SUCCESS;

```

```

80970|     IoCompleteRequest(Irp,IO_DISK_INCREMENT);
80971|     return Status;
80972| }
80973| */
80974| ASSERT(DevExt->ObjectType==OBJECT_VIRTUALDISK);
80975|
80976| // if this occurs, something is wrong..
80977| if ( !VDiskNumberOfThreads ) {
80978|     Debug(DEBUG_READ | DEBUG_PROCCALL,
80979|         | ("PSManWriteVDisk: No threads to handle
80980|         | request!!!\n"));
80981|     Irp->IoStatus.Information =
80982|         | INFORMATION_FOR_NOT_ACTIVE(Irp);
80983|     Status = VOLUME_NOT_ACTIVE_ERROR_CODE;
80984|     IoCompleteRequest(Irp,IO_NO_INCREMENT);
80985|     return Status;
80986| }
80987|
80988| if ( (gVDiskIOHandling &
80989|     | PSM_VDISK_FLAG_TACITLY_SUCCEED_WRITES) ||
80990|     | (!gVDiskDoVirtualIO) ) {
80991|     Irp->IoStatus.Information =
80992|         | IoGetCurrentIrpStackLocation(Irp)->Parameters.Write.Leng
80993|         | th;
80994|     Status = Irp->IoStatus.Status = STATUS_SUCCESS;
80995|     IoCompleteRequest(Irp, 0);
80996|     return Status;
80997| }
80998|
80999| #if 0
81000| // remmed out until we get read only snapshots
81001| | working
81002| // this will keep them from doing lost -delayed
81003| | writes.
81004| if (
81005|     | ((pPersistentDictionary)DevExt->SnapShot->Dictionary)->G
81006|     | etSnapShotFlags() & PSM_SS_FLAG_READONLY_PERSISTENT ) {
81007|     if (
81008|         | NtFsIsSystemFile(DeviceObject,Irp)==0xffffffff ) {
81009|         // if write from user application.
81010|         Status = STATUS_MEDIA_WRITE_PROTECTED;
81011|         Irp->IoStatus.Information = 0;
81012|         Irp->IoStatus.Status = Status;
81013|         IoCompleteRequest(Irp,IO_NO_INCREMENT);
81014|         return Status;
81015|     }
81016| }
81017| #endif
81018|
81019| // nothing to do... just return success.

```



```

81008|   if ( currentIrpStack->Parameters.Write.Length == 0
      | ) {
81009|       Irp->IoStatus.Information = 0;
81010|       Status = STATUS_SUCCESS;
81011|       IoCompleteRequest(Irp,IO_NO_INCREMENT);
81012|       return Status;
81013|   }
81014|
81015|   WriteRequest = (tWriteRequest
      | *)MemAllocatePoolWithTag(NonPagedPool,
      | sizeof(tWriteRequest),WRITEREQUESTTAG);
81016|   if ( WriteRequest ) {
81017|       IoMarkIrpPending(Irp);
81018|
81019|       | FillInWriteRequest(WriteRequest,DeviceObject,Irp,DevExt-
      | >BPS);
81020|
81021|       Status = Irp->IoStatus.Status   =
      | STATUS_PENDING;
81022|       Irp->IoStatus.Information = 0;
81023|
81024|       // add to queue...
81025|       ExInterlockedInsertTailList ( &WriteVDiskQueue,
81026|       | &(WriteRequest->ListEntry),
81027|       | &WriteVDiskSpinLock);
81028|
81029|
81030|       pmSignalSemaphore( &WriteVDiskSemaphore);
81031|
81032|   } else {
81033|       Debug(DEBUG_WRITE,("VDisk: Write: Out of memory
      | for write command\n"));
81034|       Status=STATUS_INSUFFICIENT_RESOURCES;
81035|       Irp->IoStatus.Information = 0;
81036|       IoCompleteRequest(Irp,IO_NO_INCREMENT);
81037|   }
81038|
81039|   //Debug(DEBUG_WRITE | DEBUG_PROCCALL,
      | ("PSManWriteVDisk Done %08x\n",Status));
81040|   return Status;
81041| }
81042|
81043| /*-----
      | -----*/
81044| STATIC NTSTATUS AddSectorsToMemoryCache (
81045|     | PVDISK_EXTENSION   DevExt,

```

```

81046|          ULARGE_INTEGER
      | Sector,
81047|          ULONG
      | Count,
81048|          char
      | *Buffer )
81049| {
81050| #ifndef _NO_WRITE_ROUTINES_
81051|     NTSTATUS Status=STATUS_SUCCESS;
81052|     ULARGE_INTEGER DS;
81053|     ULONG CountDid;
81054|     tDictSiblingInfo SI={0};
81055|
81056|     PAGED_CODE();
81057|
81058|     DS.QuadPart = (unsigned __int64)Count *
      | DevExt->BPS;
81059|     ASSERT(DevExt->ObjectType==OBJECT_VIRTUALDISK);
81060|
81061|     Status =
      | DevExt->SnapShot->Dictionary->searchAndInsertMultiple(
81062|     | (PFILTERED_EXTENSION)GetDeviceExtension(DevExt->PSMDevic
      | e),
81063|     | Sector,
81064|     | Count,
81065|     | CountDid,
81066|     | NULL,
81067|     | DS,
81068|     | Buffer,
81069|     | &SI,
81070|     | DICT_FLAG_VIRTUAL_IO);
81071|
81072|     return Status;
81073| #else
81074|     return STATUS_SUCCESS;
81075| #endif
81076| }
81077|
81078| /*-----
      | -----*/
81079| NTSTATUS InitVDiskWriteCache( ULONG WriteCacheMaxSize )

```

```

81080| {
81081| #ifndef _NO_WRITE_ROUTINES_
81082|     NTSTATUS Status=STATUS_SUCCESS;
81083|
81084|     NOT_REFERENCED(WriteCacheMaxSize);
81085|     PAGED_CODE();
81086|
81087|     return Status;
81088| #else
81089|     return STATUS_SUCCESS;
81090| #endif
81091| }
81092|
81093| /*-----
| -----*/
81094| NTSTATUS DelInitVDiskWriteCache( )
81095| {
81096| #ifndef _NO_WRITE_ROUTINES_
81097|
81098|     PAGED_CODE();
81099|
81100| #endif
81101|     return STATUS_SUCCESS;
81102| }
81103|
81104|
81105| /*-----
| -----*/
81106| STATIC NTSTATUS
81107| PSMANWriteFSObject(
81108|     IN PDEVICE_OBJECT DeviceObject,
81109|     IN PIRP Irp
81110| )
81111|
81112| /*++
81113|
81114| Routine Description:
81115|
81116|     This is the driver entry point for read requests
81117|     to disks to which the PSMAN driver has attached.
81118|     This driver collects statistics and then sets a
81119|     completion
81120|     routine so that it can collect additional
81121|     information when
81122|     the request completes. Then it calls the next
81123|     driver below
81124|     it.
81125|
81126| Arguments:
81127|

```

```

81125| DeviceObject
81126| Irp
81127|
81128| Return Value:
81129|
81130| NTSTATUS
81131|
81132| --*/
81133|
81134| {
81135|     NTSTATUS Status=STATUS_INVALID_PARAMETER;
81136|     NOT_REFERENCED(DeviceObject);
81137|
81138|     Debug(DEBUG_PROCCALL,("PSManWriteFSObject
      | Called\n"));
81139|     Irp->IoStatus.Status = Status;
81140|     Irp->IoStatus.Information = 0;
81141|
81142|     IoCompleteRequest(Irp, IO_NO_INCREMENT);
81143|     Debug(DEBUG_PROCCALL,("PSManWriteFSObject
      | Done\n"));
81144|     return Status;
81145| } // PSManWriteFSObject
81146|
81147|
81148| /*-----
      | -----*/
81149| STATIC NTSTATUS
81150| PSManWriteFSFilter(
81151|     IN PDEVICE_OBJECT DeviceObject,
81152|     IN PIRP Irp
81153| )
81154|
81155| /*++
81156|
81157| Routine Description:
81158|
81159|     Pass irp to handler
81160|
81161| Arguments:
81162|
81163|     DriverObject - Pointer to device object to being
      | shutdown by system.
81164|     Irp          - IRP involved.
81165|
81166| Return Value:
81167|
81168|     NT Status
81169|
81170| --*/

```

```

81171|
81172| {
81173|     return SfFsWrite(DeviceObject,Irp);
81174| } // end PSMANWRITEFSFILTER()
81175|
81176|
81177|
81178| File Listing: WRITE.h
81179|
81180| NTSTATUS
81181| PSMANWRITE(
81182|     IN PDEVICE_OBJECT DeviceObject,
81183|     IN PIRP Irp
81184| );
81185|
81186| STATIC NTSTATUS
81187| PSMANWRITEOBJECT(
81188|     IN PDEVICE_OBJECT DeviceObject,
81189|     IN PIRP Irp
81190| );
81191|
81192| STATIC NTSTATUS
81193| PSMANWRITEDEVICE(
81194|     IN PDEVICE_OBJECT DeviceObject,
81195|     IN PIRP Irp
81196| );
81197|
81198| STATIC NTSTATUS
81199| PSMANWRITEVDISK(
81200|     IN PDEVICE_OBJECT DeviceObject,
81201|     IN PIRP Irp
81202| );
81203|
81204| STATIC NTSTATUS
81205| PSMANWRITEFSFILTER(
81206|     IN PDEVICE_OBJECT DeviceObject,
81207|     IN PIRP Irp
81208| );
81209|
81210| STATIC NTSTATUS
81211| PSMANWRITEFSOBJECT(
81212|     IN PDEVICE_OBJECT DeviceObject,
81213|     IN PIRP Irp
81214| );
81215|
81216| void WriteVDiskThread ( PVOID Context );
81217| NTSTATUS DelInitVDiskWriteCache(void);
81218| NTSTATUS InitVDiskWriteCache( ULONG WriteCacheMaxSize
81219|     | );

```

```

81220|
81221| NTSTATUS InsertFileObjectToSkip ( PFILE_OBJECT
      | FileObject );
81222| NTSTATUS DeleteFileObjectToSkip ( PFILE_OBJECT
      | FileObject );
81223| NTSTATUS IsFileObjectToSkip ( PFILE_OBJECT FileObject
      | );
81224| NTSTATUS InitWriteModule(void);
81225| BOOLEAN IsCacheFile( PDEVICE_OBJECT DeviceObject, PIRP
      | Irp );
81226|
81227|
81228|
81229| PSM to OTM Client Bridge Source
81230|
81231| The OTM Bridge allows the PSM system to support an OTM
      | application level client. --LPW
81232|
81233|
81234|
81235| File Listing: buildnum_otmbridge.h
81236|
81237| // buildnum_otmbridge.h - Determines build number of
      | OTM Bridge Driver.
81238|
81239| #define _OtmBridge_BuildNumber_      102
81240| #define _OtmBridge_BuildNumberStr_   "102\0"
81241| #define _OtmBridge_BuildNumberWStr_  L"102\0"
81242|
81243|
81244|
81245| File Listing: CDP.h
81246|
81247| #define VER_COMPANYNAME_STR          "Columbia Data
      | Products, Inc."
81248| #define VER_LEGALTRADEMARKS_STR      "PSM\256 is a
      | trademark of " VER_COMPANYNAME_STR
81249|
81250| #define VER_LEGALCOPYRIGHT_YEARS     "1995-2001"
81251| #define VER_LEGALCOPYRIGHT_STR       "Copyright \251 "
      | VER_LEGALCOPYRIGHT_YEARS " " VER_COMPANYNAME_STR
81252|
81253| #if DBG
81254| #define VER_DEBUG                     VS_FF_DEBUG
81255| #else
81256| #define VER_DEBUG                     0
81257| #endif
81258|
81259| #if BETA
81260| #define VER_PRODUCTBETA_STR           "BETA"

```

```

81261| #define VER_PRERELEASE      VS_FF_PRERELEASE
81262| #else
81263| #define VER_PRERELEASE      0
81264| #define VER_PRODUCTBETA_STR  ""
81265| #endif
81266|
81267| #define VER_FILEFLAGSMASK     VS_FFI_FILEFLAGSMASK
81268| #define VER_FILEOS           VOS_NT_WINDOWS32
81269| #define VER_FILEFLAGS        (VER_PRERELEASE |
    | VER_DEBUG)
81270|
81271|
81272|
81273| File Listing: init.cpp
81274|
81275| #include "precomp.h"
81276|
81277| extern "C" NTSTATUS BridgeFilterOtm( void );
81278|
81279| #pragma alloc_text(INIT, DriverEntry)
81280| #pragma alloc_text(INIT, BridgeFilterOtm)
81281|
81282| PDEVICE_OBJECT BridgeControlObject=NULL;
81283| PDRIVER_OBJECT BridgeDriverObject=NULL;
81284| PDEVICE_OBJECT OtmFilteredObject=NULL;
81285| PDEVICE_OBJECT PsmDeviceObject=NULL;
81286| PFILE_OBJECT PsmFileObject=NULL;
81287| WCHAR          gRegistryPath[256]={0};
81288| ULONG          PsmBuildNumber=0;
81289| ULONG          PsmMajorVersion=0;
81290| ULONG          PsmMinorVersion=0;
81291|
81292|
81293| extern "C"
81294| NTSTATUS
81295| DriverEntry(
81296|     IN PDRIVER_OBJECT DriverObject,
81297|     IN PUNICODE_STRING RegistryPath
81298| )
81299| {
81300| #ifdef DEBUG
81301|     ULONG shouldBreak=0;
81302|
81303|     | Reg_GetULONGKey(RegistryPath->Buffer,L"BreakOnEntry",0,&
    | shouldBreak);
81304|
81305|     if ( shouldBreak ) {
81306|         DbgPrint("Bridge: Breaking in driver entry\n");
81307|         DbgBreakPoint();

```

```

81308|    }
81309| #endif
81310|
81311|    UNICODE_STRING deviceNameUnicodeString={0};
81312|    UNICODE_STRING deviceLinkUnicodeString={0};
81313|    NTSTATUS Status=STATUS_SUCCESS;
81314|
81315|    BridgeDriverObject = DriverObject;
81316|
81317|
81318|    | RtlCopyMemory(gRegistryPath,RegistryPath->Buffer,Registr
81319|    | yPath->Length);
81320|
81321|    RtlInitUnicodeString
81322|    | (&deviceNameUnicodeString,BRIDGE_DEVICE_NAME);
81323|
81324|    RtlInitUnicodeString
81325|    | (&deviceLinkUnicodeString,BRIDGE_LINK_NAME);
81326|
81327|    Status = IoCreateDevice (   DriverObject,
81328|    | sizeof(BRIDGE_DEVICE_EXTENSION),
81329|    | &deviceNameUnicodeString,
81330|    | FILE_DEVICE_UNKNOWN,
81331|    | FILE_DEVICE_SECURE_OPEN,
81332|    | FALSE,
81333|    | &BridgeControlObject
81334|    | );
81335|
81336|    if(NT_SUCCESS(Status)) {
81337|        PBRIDGE_DEVICE_EXTENSION DevExt =
81338|        | (PBRIDGE_DEVICE_EXTENSION)BridgeControlObject->DeviceExt
81339|        | ension;
81340|
81341|        DevExt->DeviceObject = BridgeControlObject;
81342|        DevExt->DriverObject = DriverObject;
81343|        DevExt->ObjectType = OBJECT_INTERNAL;
81344|
81345|        DriverObject->MajorFunction[IRP_MJ_CREATE]
81346|        | = BridgeCreate;
81347|        DriverObject->MajorFunction[IRP_MJ_CLOSE]
81348|        | = BridgeClose;
81349|
81350|        | DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]
81351|        | = BridgeDeviceControl;
81352|
81353|        DriverObject->MajorFunction[IRP_MJ_CLEANUP]
81354|        | = BridgeCleanup;
81355|
81356|        DriverObject->DriverUnload

```



```

    | = BridgeUnload;
81344|
81345|     // Create a symbolic link, e.g. a name that a
    | Win32 app can specify
81346|     // to open the device
81347|
81348|     Status = IoCreateSymbolicLink (
    | &deviceLinkUnicodeString, &deviceNameUnicodeString);
81349|
81350|     if(NT_SUCCESS(Status)) {
81351|         NTSTATUS FilterStatus = BridgeFilterOtm();
81352|
81353|         if(NT_SUCCESS(FilterStatus)) {
81354|             Debug(DEBUG_INIT,("Bridge: Success
    | bridging OTM to PSM\n"));
81355|         } else {
81356|             Debug(DEBUG_INIT,("Bridge: Error %08x
    | bridging OTM to PSM\n",FilterStatus));
81357|             IoDeleteSymbolicLink
    | (&deviceLinkUnicodeString );
81358|             IoDeleteDevice(BridgeControlObject);
81359|             BridgeControlObject=NULL;
81360|         }
81361|         // We dont want the system to report a
    | failure if OTM or PSM is not installed
81362|         // so we return success. This means we
    | dont load, but i dont know of any other
81363|         // way of getting Windows 2000 from issuing
    | that nasty event message.
81364|         Status = STATUS_SUCCESS;
81365|     } else {
81366|         IoDeleteDevice(BridgeControlObject);
81367|         BridgeControlObject=NULL;
81368|     }
81369| }
81370|
81371|
81372| return Status;
81373| }
81374|
81375| extern "C"
81376| NTSTATUS BridgeFilterOtm( void )
81377| {
81378|     PDEVICE_OBJECT MyDevObj;
81379|     PFILE_OBJECT FileObject;
81380|     PDEVICE_OBJECT OtmDevObj;
81381|     UNICODE_STRING Uni;
81382|     NTSTATUS Status;
81383|     POTM_FILTER_EXTENSION DevExt;
81384|

```

```

81385|    // lets see if psm is installed, otherwise there is
      | nothing for us to do
81386|    RtlInitUnicodeString(&Uni,
      | L"\\Device\\PSMan_0200");
81387|
81388|    Status = IoGetDeviceObjectPointer( &Uni,
81389|
      | FILE_READ_ATTRIBUTES,
81390|
      | &PsmFileObject,
81391|
      | &PsmDeviceObject);
81392|    if(NT_SUCCESS(Status)) {
81393|        tPSM_VersionInfo Version;
81394|
81395|        PsmGetVersion( &Version );
81396|
81397|        PsmBuildNumber = (Version.Version & 0xffff0000)
      | >> 16;
81398|        PsmMajorVersion = (Version.Version &
      | 0x0000ff00) >> 8;
81399|        PsmMinorVersion = (Version.Version &
      | 0x000000ff);
81400|
81401|        // dont support PSM 1.x (which has a version of
      | 2.00 and a build number less than 200 decimal)
81402|        if(PsmBuildNumber>=200) {
81403|            Debug(DEBUG_INIT,("Bridge: PSM v%d.%02x
      | build %d is
      | installed\n",PsmMajorVersion,PsmMinorVersion,PsmBuildNum
      | ber));
81404|
81405|            // lets see if otm is there so we can
      | filter it.
81406|            RtlInitUnicodeString(&Uni,
      | L"\\Device\\OtmMan_0110");
81407|
81408|            Status = IoGetDeviceObjectPointer( &Uni,
81409|
      | FILE_READ_ATTRIBUTES,
81410|
      | &FileObject,
81411|
      | &OtmDevObj);
81412|            if(NT_SUCCESS(Status)) {
81413|                ULONG SizeOfDeviceExt =
      | sizeof(OTM_FILTER_EXTENSION);
81414|
81415|                Debug(DEBUG_INIT,("Bridge: OTM v1.1x is
      | installed\n"));
81416|

```

```

81417|         Status =
      | IoCreateDevice(BridgeDriverObject,
81418|         | SizeOfDeviceExt,
81419|         NULL,
81420|         | FILE_DEVICE_UNKNOWN,
81421|         | FILE_DEVICE_SECURE_OPEN,
81422|         FALSE,
81423|         &MyDevObj);
81424|
81425|         if ( NT_SUCCESS(Status) ) {
81426|             Debug(DEBUG_INIT,("Bridge: Filtered
      | Device %p created for driver OTM\n",MyDevObj));
81427|
81428|             DevExt =
      | (POTM_FILTER_EXTENSION)GetDeviceExtension(MyDevObj);
81429|             ASSERT(DevExt);
81430|
81431|             DevExt->DeviceObject = MyDevObj;
81432|             DevExt->DriverObject =
      | BridgeDriverObject;
81433|             DevExt->ObjectType =
      | OBJECT_OTM_FILTER;
81434|
81435|             Status = IoAttachDevice(MyDevObj,
81436|             &Uni,
81437|             | &DevExt->TargetDeviceObject);
81438|
81439|             if ( NT_SUCCESS(Status) ) {
81440|
81441|                 //
81442|                 // Propagate driver's alignment
      | requirements.
81443|                 //
81444|
81445|                 MyDevObj->Flags
      | = DevExt->TargetDeviceObject->Flags;
81446|                 MyDevObj->AlignmentRequirement
      | = DevExt->TargetDeviceObject->AlignmentRequirement;
81447|                 MyDevObj->StackSize
      | = DevExt->TargetDeviceObject->StackSize+1;
81448|                 MyDevObj->DeviceType
      | = DevExt->TargetDeviceObject->DeviceType;
81449|                 MyDevObj->Characteristics
      | = DevExt->TargetDeviceObject->Characteristics;
81450|
81451|                 /*lint -save -e613 */

```

```

81452|             MyDevObj->Flags &=
      | ~DO_DEVICE_INITIALIZING;
81453|             /*lint -restore */
81454|
81455|             // we are now filtered over the
      | otman device object
81456|             OtmFilteredObject = MyDevObj;
81457|
81458|             /*lint -save -e740 */
81459|
      | LogError((PDEVICE_OBJECT)BridgeDriverObject,NULL,BRIDGE_
      | ACTIVATED,0,NULL,0,NULL,0);
81460|             /*lint -restore */
81461|
81462|             } else {
81463|                 Debug(DEBUG_INIT,("Bridge:
      | Error! %08x Unable to attach to otm\n",Status));
81464|                 IoDeleteDevice(MyDevObj);
81465|             }
81466|
81467|             } else {
81468|                 Debug(DEBUG_INIT,("Bridge: Error!
      | %08x Unable to create device for otm\n",Status));
81469|             }
81470|
81471|             // we are now done with deviceobject
      | (always free fileobject when given both)
81472|             ObDereferenceObject(FileObject);
81473|             FileObject = NULL;
81474|             } else {
81475|                 Debug(DEBUG_INIT,("Bridge: OTM v1.1x is
      | not installed\n"));
81476|                 Status = STATUS_NOT_FOUND;
81477|             }
81478|
81479|             } else {
81480|                 Debug(DEBUG_INIT,("Bridge: PSM 1.x build %d
      | not supported\n",PsmBuildNumber));
81481|
81482|             /*lint -save -e740 */
81483|
      | LogError((PDEVICE_OBJECT)BridgeDriverObject,NULL,BRIDGE_
      | WRONG_PSM_VERSION,BRIDGE_WRONG_PSM_VERSION,NULL,0,NULL,0
      | );
81484|             /*lint -restore */
81485|
81486|             Status = STATUS_NOT_FOUND;
81487|             }
81488|
81489|             // if error, free psm

```

```

81490|     if(!NT_SUCCESS(Status)) {
81491|         ObDereferenceObject(PsmFileObject);
81492|         PsmFileObject = NULL;
81493|         PsmDeviceObject=NULL;
81494|     }
81495| } else {
81496|     Debug(DEBUG_INIT,("Bridge: PSM v2.x is not
| installed\n"));
81497|     Status = STATUS_NOT_FOUND;
81498|     PsmFileObject = NULL;
81499|     PsmDeviceObject=NULL;
81500| }
81501|
81502| return Status;
81503| }
81504|
81505| // -----
81506| VOID
81507| BridgeUnload(
81508|     IN PDRIVER_OBJECT DriverObject
81509| )
81510| {
81511|     Debug(DEBUG_OTM_FILTER,("Bridge: Unload\n"));
81512|
81513|     if(BridgeControlObject) {
81514|         UNICODE_STRING deviceLinkUnicodeString={0};
81515|
81516|         Debug(DEBUG_OTM_FILTER,("Bridge: Unload:
| Removing Control object\n"));
81517|
81518|         RtlInitUnicodeString
| (&deviceLinkUnicodeString,BRIDGE_LINK_NAME);
81519|         IoDeleteSymbolicLink (&deviceLinkUnicodeString
| );
81520|
81521|         IoDeleteDevice(BridgeControlObject);
81522|         BridgeControlObject=NULL;
81523|     }
81524|
81525|     if(OtmFilteredObject) {
81526|         POTM_FILTER_EXTENSION
| DevExt=(POTM_FILTER_EXTENSION)GetDeviceExtension(OtmFilt
| eredObject);
81527|
81528|         Debug(DEBUG_OTM_FILTER,("Bridge: Unload:
| Detaching from OTM\n"));
81529|         IoDetachDevice(DevExt->TargetDeviceObject);
81530|         IoDeleteDevice(OtmFilteredObject);
81531|         OtmFilteredObject=NULL;
81532|     }

```

```

81533|
81534|     if(PsmFileObject) {
81535|         Debug(DEBUG_OTM_FILTER,("Bridge: Unload:
      | Detaching from PSM\n"));
81536|
81537|         // get rid of reference count to psm
81538|         ObDereferenceObject(PsmFileObject);
81539|
81540|         PsmFileObject = NULL;
81541|         PsmDeviceObject=NULL;
81542|
81543|         // only log the event if we actually filtered
      | anything
81544|         // ie, we are still loaded if otm or psm is not
      | installed
81545|         // but we didnt generate any event, so if we
      | are being unloaded
81546|         // lets make sure to be silent
81547|
81548|         /*lint -save -e740 */
81549|
      | LogError((PDEVICE_OBJECT)DriverObject,NULL,BRIDGE_DEACTI
      | VATED,0,NULL,0,NULL,0);
81550|         /*lint -restore */
81551|     }
81552|
81553|
81554|     return;
81555| }
81556|
81557| // -----
81558| NTSTATUS
81559| BridgeCreate (
81560|     IN PDEVICE_OBJECT DeviceObject,
81561|     IN PIRP Irp
81562| )
81563| {
81564|     POTM_FILTER_EXTENSION
      | DevExt=(POTM_FILTER_EXTENSION)GetDeviceExtension(DeviceO
      | bject);
81565|
81566|     Debug(DEBUG_OTM_FILTER,("Bridge: Create: %08x
      | (%d)-%08x\n",DeviceObject,DevExt->ObjectType,Irp));
81567|
81568|     if(DevExt->ObjectType == OBJECT_INTERNAL) {
81569|         Irp->IoStatus.Status = STATUS_SUCCESS;
81570|         Irp->IoStatus.Information = 0 ;
81571|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
81572|         return STATUS_SUCCESS;
81573|     }

```

```

81574|
81575|  ASSERT(DevExt->ObjectType == OBJECT_OTM_FILTER);
81576|
81577|  // call directly into psm
81578|  ASSERT(PsmDeviceObject);
81579|  ASSERT(PsmDeviceObject->DriverObject);
81580|
81581|  | ASSERT(PsmDeviceObject->DriverObject->MajorFunction[IRP_
81582|  | MJ_CREATE]);
81583| }
81584|
81585| // -----
81586| NTSTATUS
81587| BridgeClose (
81588|  IN PDEVICE_OBJECT DeviceObject,
81589|  IN PIRP Irp
81590| )
81591| {
81592|  POTM_FILTER_EXTENSION
81593|  | DevExt=(POTM_FILTER_EXTENSION)GetDeviceExtension(DeviceO
81594|  | bject);
81595|
81596|  Debug(DEBUG_OTM_FILTER,("Bridge: Close: %08x
81597|  | (%d)-%08x\n", DeviceObject, DevExt->ObjectType, Irp));
81598|
81599|  if(DevExt->ObjectType == OBJECT_INTERNAL) {
81600|      Irp->IoStatus.Status = STATUS_SUCCESS;
81601|      Irp->IoStatus.Information = 0 ;
81602|      IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
81603|      return STATUS_SUCCESS;
81604|  }
81605|
81606|  ASSERT(DevExt->ObjectType == OBJECT_OTM_FILTER);
81607|
81608|  // call directly into psm
81609|  ASSERT(PsmDeviceObject);
81610|  ASSERT(PsmDeviceObject->DriverObject);
81611|
81612|  | ASSERT(PsmDeviceObject->DriverObject->MajorFunction[IRP_
81613|  | MJ_CLOSE]);
81614|
81615|  return
81616|  | PsmDeviceObject->DriverObject->MajorFunction[IRP_MJ_CLOS
81617|  | E](PsmDeviceObject, Irp);
81618| }
81619|
81620|

```

```

81613| // -----
81614| NTSTATUS
81615| BridgeCleanup (
81616|     IN PDEVICE_OBJECT DeviceObject,
81617|     IN PIRP Irp
81618| )
81619| {
81620|     POTM_FILTER_EXTENSION
        | DevExt=(POTM_FILTER_EXTENSION)GetDeviceExtension(DeviceO
        | bject);
81621|
81622|     Debug(DEBUG_OTM_FILTER,("Bridge: Cleanup: %08x
        | (%d)-%08x\n", DeviceObject, DevExt->ObjectType, Irp));
81623|
81624|     if(DevExt->ObjectType == OBJECT_INTERNAL) {
81625|         Irp->IoStatus.Status = STATUS_SUCCESS;
81626|         Irp->IoStatus.Information = 0 ;
81627|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
81628|         return STATUS_SUCCESS;
81629|     }
81630|
81631|     ASSERT(DevExt->ObjectType == OBJECT_OTM_FILTER);
81632|
81633|     // call directly into psm
81634|     ASSERT(PsmDeviceObject);
81635|     ASSERT(PsmDeviceObject->DriverObject);
81636|
        | ASSERT(PsmDeviceObject->DriverObject->MajorFunction[IRP_
        | MJ_CLEANUP]);
81637|
81638|     return
        | PsmDeviceObject->DriverObject->MajorFunction[IRP_MJ_CLEA
        | NUP](PsmDeviceObject, Irp);
81639| }
81640|
81641| /*--- end of file init.cpp ---*/
81642|
81643|
81644|
81645| File Listing: init.h
81646|
81647| #define OBJECT_INTERNAL    0
81648| #define OBJECT_OTM_FILTER  1
81649|
81650| typedef struct _BRIDGE_DEVICE_EXTENSION {
81651|     // common header
81652|     PDEVICE_OBJECT DeviceObject;        // Back
        | pointer to device object
81653|     PDIVER_OBJECT DriverObject;        // The
        | driver object for use on repartitioning.

```



```

81654|  ULONG      ObjectType;
81655|
81656|  // Extension specific
81657| } BRIDGE_DEVICE_EXTENSION, *PBRIDGE_DEVICE_EXTENSION;
81658|
81659| typedef struct _OTM_FILTER_EXTENSION {
81660|  // common header
81661|  PDEVICE_OBJECT DeviceObject;      // Back
      | pointer to device object
81662|  PDRIVER_OBJECT DriverObject;      // The
      | driver object for use on repartitioning.
81663|  ULONG      ObjectType;
81664|
81665|  // Extension specific
81666|  PDEVICE_OBJECT TargetDeviceObject;
81667| } OTM_FILTER_EXTENSION, *POTM_FILTER_EXTENSION;
81668|
81669|
81670| #define BRIDGE_DEVICE_NAME  L"\\Device\\OtmBridge"
81671| #define BRIDGE_LINK_NAME   L"\\DosDevices\\OtmBridge"
81672|
81673| #define DWORD_ALIGN(x)      (((x)+31)/32)*32
81674|
81675| extern "C" {
81676|  NTSTATUS DriverEntry(
81677|      IN PDRIVER_OBJECT DriverObject,
81678|      IN PUNICODE_STRING RegistryPath
81679|  );
81680|
81681|  NTSTATUS BridgeCreate (
81682|      IN PDEVICE_OBJECT DeviceObject,
81683|      IN PIRP Irp
81684|  );
81685|
81686|  NTSTATUS BridgeClose (
81687|      IN PDEVICE_OBJECT DeviceObject,
81688|      IN PIRP Irp
81689|  );
81690|
81691|  NTSTATUS BridgeCleanup (
81692|      IN PDEVICE_OBJECT DeviceObject,
81693|      IN PIRP Irp
81694|  );
81695|
81696|  VOID BridgeUnload(
81697|      IN PDRIVER_OBJECT DriverObject
81698|  );
81699|
81700|  NTSTATUS BridgeDeviceControl (
81701|      IN PDEVICE_OBJECT DeviceObject,

```

```

81702|     IN PIRP Irp
81703| );
81704|
81705|
81706| }
81707|
81708|
81709| #ifdef DEBUG
81710| #define STATIC
81711| #define Debug(a,x) DbgPrint x
81712| #else
81713| #define Debug(a,x)
81714| #define STATIC static
81715| #endif
81716|
81717| #define GetDeviceExtension(x) x->DeviceExtension
81718|
81719| #define INVALID_HANDLE_VALUE ((PVOID)-1)
81720| #define IsValidHandle(Handle) (((Handle)!= (void*)0) &&
    | ((Handle)!= (void*)-1))
81721|
81722| #define pmWaitForSingleObject(o,t)
    | KeWaitForSingleObject(o,Suspended,(KPROCESSOR_MODE)Kerne
    | IMode,FALSE,t)
81723|
81724| extern PDEVICE_OBJECT BridgeControlObject;
81725| extern PDRIVER_OBJECT BridgeDriverObject;
81726| extern PDEVICE_OBJECT PsmDeviceObject;
81727| extern ULONG PsmBuildNumber;
81728| extern ULONG PsmMajorVersion;
81729| extern ULONG PsmMinorVersion;
81730| extern WCHAR gRegistryPath[];
81731|
81732| #define OTM_BRIDGE_INSTALLED_BIT 0x8000
81733|
81734| /*--- end of file init.h ---*/
81735|
81736|
81737|
81738| File Listing: otm.cpp
81739|
81740| #include <precomp.h>
81741|
81742| #include <otm_private.h>
81743|
81744| #include <buildnum_otmbridge.h>
81745|
81746| /*-----
    | -----*/
81747| NTSTATUS OtmGetVersion(PIRP Irp)

```

```

81748| {
81749|     NTSTATUS Status = STATUS_SUCCESS;
81750|     ULONG RetSize=0;
81751|
81752|     __try {
81753|         PIO_STACK_LOCATION plrpStack =
            | IoGetCurrentIrpStackLocation (Irp) ;
81754|         tOTM_VersionInfo *Buffer=NULL;
81755|
81756|         Debug(DEBUG_PROCCALL |
            | DEBUG_OTM_FILTER,("Bridge: GetVersion\n"));
81757|         // METHOD_BUFFERED
81758|         // Irp->AssociatedIrp.SystemBuffer =
            | Input/Output buffer
81759|
81760|         if
            | (plrpStack->Parameters.DeviceIoControl.InputBufferLength
            | < sizeof(DWORD)) {
81761|             Debug(DEBUG_OTM_FILTER,("Bridge: Error!
            | Input IOCTL buffer not big enough\n"));
81762|             Status = STATUS_INVALID_BUFFER_SIZE;
81763|         } else
81764|         if
            | (plrpStack->Parameters.DeviceIoControl.OutputBufferLengt
            | h < sizeof(tOTM_VersionInfo)) {
81765|             Debug(DEBUG_OTM_FILTER,("Bridge: Error!
            | Output IOCTL buffer not big enough\n"));
81766|             Status = STATUS_INVALID_BUFFER_SIZE;
81767|         } else {
81768|
81769|             Buffer = (tOTM_VersionInfo
            | *)Irp->AssociatedIrp.SystemBuffer;
81770|
81771|             if (Buffer) {
81772|
81773|                 // inbound is a DWORD size
81774|                 // we are doing this on the fact that
            | the input and output buffers
81775|                 // are the same and that Size is the
            | first argument of the output.
81776|
81777|                 | if(Buffer->Size==sizeof(tOTM_VersionInfo)) {
81778|                     // Set our build number, with the
            | version we our emulating
81779|                     Buffer->Version          =
            | OTM_BRIDGE_INSTALLED_BIT | (_OtmBridge_BuildNumber_ <<
            | 16) | OTM_SUPPORTED_VERSION_NUMBER;
81780|                     Buffer->LoVersion        =
            | OTM_SUPPORTED_LOW_VERSION_NUMBER;

```

```

81781|         Buffer->OSType          = 0;
81782|         Buffer->OSVersion         = 0;
81783|         Buffer->CommunicationMethods = 0;
81784|
81785|         Status   = STATUS_SUCCESS;
81786|         RetSize  =
            | sizeof(tOTM_VersionInfo);
81787|     } else {
81788|         Status =
            | STATUS_INVALID_BUFFER_SIZE;
81789|     }
81790| } else {
81791|     Debug(DEBUG_OTM_FILTER,("Bridge: Error!
            | Buffer is NULL\n"));
81792|     Status = STATUS_INVALID_PARAMETER;
81793| }
81794| }
81795| } __except( EXCEPTION_EXECUTE_HANDLER ) {
81796|     Status = GetExceptionCode();
81797|     Debug(DEBUG_OTM_FILTER,("Bridge: OtmGetVersion:
            | Error! Exception %08x\n", Status));
81798| }
81799|
81800| Irp->IoStatus.Information = RetSize;
81801| Irp->IoStatus.Status = Status;
81802| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
81803| return Status;
81804| }
81805|
81806| typedef struct sOtmToPsm {
81807|     PIRP Irp;
81808|     PVOID OtmOut;
81809|     PVOID PsmOut;
81810|     IO_STATUS_BLOCK IoStatus;
81811|     PKEVENT Event;
81812| } tOtmToPsm;
81813|
81814| VOID
81815| OtmToPsmCreateSnapShotThread(
81816|     IN PVOID Context
81817| )
81818| {
81819|     tOtmToPsm *OtmToPsm=(tOtmToPsm*)Context;
81820|     tOpenTransactionOutInternal
            | *PsmOut=(tOpenTransactionOutInternal
            | *)OtmToPsm->PsmOut;
81821|     tOTMOpenTransactionInInternal
            | *OtmIn=(tOTMOpenTransactionInInternal
            | *)OtmToPsm->OtmOut;
81822|     tOTMOpenTransactionOutInternal

```

```

    | *OtmOut=(tOTMOpenTransactionOutInternal
    | *)OtmToPsm->OtmOut;
81823|
81824|    // wait for open to finish
81825|    Debug(DEBUG_OTM_FILTER,("Bridge: CreateCallBack:
    | Waiting for PSM open to complete\n"));
81826|
81827|    | KeWaitForSingleObject(OtmToPsm->Event,Suspended,(KPROCESS
    | SOR_MODE)KernelMode,FALSE,NULL);
81828|
81829|    ULONG Status = OtmToPsm->IoStatus.Status;
81830|    ULONG RetSize = OtmToPsm->IoStatus.Information;
81831|
81832|    ASSERT(!PsmOut->Persistent);
81833|
81834|    Debug(DEBUG_OTM_FILTER,("Bridge: CreateCallBack:
    | Status=%08x, Info=%08x\n",Status,RetSize));
81835|
81836|    __try {
81837|        if(NT_SUCCESS(Status)) {
81838|            if(OtmIn->InternalFlags &
    | PSM_IFLAG_NEW_SNAPSHOT) {
81839|
81840|                OtmOut->KernelSnapShotPointer =
    | PsmOut->KernelSnapShotPointer;
81841|                OtmOut->SnapShotTime =
    | PsmOut->SnapShotTime;
81842|                OtmOut->Instance = PsmOut->Instance;
81843|
    | wcscpy(OtmOut->CacheFileName,L"C:\\psm");
81844|
81845|                OtmOut->NumberOfDevices =
    | PsmOut->NumberOfDevices;
81846|
81847|                WCHAR
    | *p=(WCHAR*)(OtmOut->DeviceName+PsmOut->NumberOfDevices);
81848|                for(ULONG
    | i=0;i<PsmOut->NumberOfDevices;i++) {
81849|
    | wcscpy(p,(WCHAR*)DN_MakePointer(PsmOut,PsmOut->DeviceNam
    | e[i]));
81850|                OtmOut->DeviceName[i] =
    | DN_MakeOffset(OtmOut,p);
81851|                p+=wcslen(p)+1;
81852|            }
81853|            RetSize = (char*)p-(char*)OtmOut;
81854|        }
81855|    } else {
81856|        RetSize = 0;

```

```

81857|     }
81858| } __except( EXCEPTION_EXECUTE_HANDLER ) {
81859|     Status = GetExceptionCode();
81860|     RetSize = 0;
81861|     Debug(DEBUG_OTM_FILTER,("Bridge:
      | OtmTurnOnOtmCallBack: Error! Exception %08x\n",
      | Status));
81862| }
81863|
81864| OtmToPsm->Irp->IoStatus.Status = Status;
81865| OtmToPsm->Irp->IoStatus.Information = RetSize;
81866|
81867| IoCompleteRequest(OtmToPsm->Irp,IO_NO_INCREMENT);
81868|
81869| ExFreePool(OtmToPsm->Event);
81870| ExFreePool(OtmToPsm);
81871| return;
81872| }
81873|
81874|
81875| /*-----
      | -----*/
81876| NTSTATUS OtmTurnOnOtm(PIRP Irp)
81877| {
81878| #if 1
81879|     NTSTATUS Status = STATUS_SUCCESS;
81880|
81881|     __try {
81882|         PIO_STACK_LOCATION plrpStack =
            | IoGetCurrentIrpStackLocation (Irp) ;
81883|         tOTMOpenTransactionInInternal *In=NULL;
81884|         tOTMOpenTransactionOutInternal *Out=NULL;
81885|         tOpenTransactionInInternal *PsmIn=NULL;
81886|         PIRP newIrp;
81887|         tOtmToPsm *OtmToPsm;
81888|
81889|         // METHOD_BUFFERED
81890|         // Irp->AssociatedIrp.SystemBuffer =
            | Input/Output buffer
81891|
81892|         Debug(DEBUG_PROCCALL |
            | DEBUG_OTM_FILTER,("Bridge: OtmTurnOnOtm\n"));
81893|         In = (tOTMOpenTransactionInInternal
            | *)Irp->AssociatedIrp.SystemBuffer;
81894|
81895|         if
            | (plrpStack->Parameters.DeviceIoControl.InputBufferLength
            | < sizeof(tOTMOpenTransactionInInternal)) {
81896|             Debug(DEBUG_OTM_FILTER,("Bridge:
            | OtmTurnOnOtm: Error! IOCTL buffer not big enough\n"));

```

```

81897|         Status = STATUS_INVALID_BUFFER_SIZE;
81898|     } else
81899|     if ((!In) ||
    | (In->Size!=sizeof(tOTMOpenTransactionInInternal))){
81900|         Debug(DEBUG_OTM_FILTER,("Bridge:
    | OtmTurnOnOtm: Error! Buffer is NULL or not right
    | size\n"));
81901|         Status = STATUS_INVALID_PARAMETER;
81902|     } else
81903|     if ((In->InternalFlags &
    | OTM_IFLAG_NEW_SNAPSHOT) &&
81904|
    | (pIrpStack->Parameters.DeviceIoControl.OutputBufferLengt
    | h < sizeof(tOTMOpenTransactionOutInternal))) {
81905|         Debug(DEBUG_OTM_FILTER,("Bridge:
    | OtmTurnOnOtm: Error! IOCTL buffer not big enough\n"));
81906|         Status = STATUS_INVALID_BUFFER_SIZE;
81907|     } else {
81908|
81909|
81910|         // okay, to make life simple for us we will
    | allocate another irp
81911|         // and send this to psm with the correct
    | parameters. We will have
81912|         // a completion routine that will copy from
    | the new to the old
81913|         // and complete the original irp. We have
    | to do this because this
81914|         // api executes asynchronously.
81915|         OtmToPsm=
    | (tOtmToPsm*)ExAllocatePoolWithTag(PagedPool,65536+sizeof
    | (tOtmToPsm),'PmtO');
81916|         Psmln = (tOpenTransactionInInternal
    | *) (OtmToPsm+1);
81917|         if(OtmToPsm) {
81918|
81919|             OtmToPsm->Event =
    | (PKEVENT)ExAllocatePoolWithTag(NonPagedPool,sizeof(KEVEN
    | T),'EmtO');
81920|             if(OtmToPsm->Event) {
81921|                 KeInitializeEvent( OtmToPsm->Event,
    | NotificationEvent, FALSE);
81922|
81923|                 OtmToPsm->Irp = Irp;
81924|                 OtmToPsm->OtmOut =
    | Irp->AssociatedIrp.SystemBuffer;
81925|                 OtmToPsm->PsmOut = Psmln;
81926|                 OtmToPsm->IoStatus.Status =
    | STATUS_PENDING;
81927|                 OtmToPsm->IoStatus.Information = 0;

```

```

81928|
81929|         PsmIn->Size =
      | sizeof(tOpenTransactionInternal);
81930|         PsmIn->Flags = 0;
81931|         PsmIn->InternalFlags =
      | PSM_IFLAG_NEW_SNAPSHOT;
81932|         PsmIn->AbortEvent = In->AbortEvent;
81933|         PsmIn->ErrorEvent = In->ErrorEvent;
81934|         PsmIn->SnapShotFlags = 0;
81935|
81936|         if(PsmMajorVersion==2) {
81937|             if(PsmMinorVersion==0) {
81938|
      | Debug(DEBUG_OTM_FILTER,("Bridge: CreateSnapShot: PSM
      | version 2.0 installed, creating persistent
      | snapshot\n"));
81939|                 // version 2.0 doesnt
      | support temporary snapshots
81940|                 // so we just say its
      | persistent...
81941|
      | PsmIn->SnapShotFlags=PSM_2_00_SS_FLAG_READWRITE;
81942|
      | PsmIn->InternalFlags|=PSM_IFLAG_PERSISTENT;
81943|             } else {
81944|
      | Debug(DEBUG_OTM_FILTER,("Bridge: CreateSnapShot: PSM
      | version 2.1+ installed, creating temporary
      | snapshot\n"));
81945|                 // version 2.1 or above
      | support temporary snapshots
81946|
      | PsmIn->SnapShotFlags=PSM_SS_FLAG_T_READWRITE;
81947|             }
81948|
      |
81949|             } else {
81950|                 // dont support version 1, init
      | should have not even loaded
81951|                 ASSERT(PsmMajorVersion>2);
81952|
      | Debug(DEBUG_OTM_FILTER,("Bridge: CreateSnapShot: PSM
      | version 3.0+ installed, creating temporary
      | snapshot\n"));
81953|
      | PsmIn->SnapShotFlags=PSM_SS_FLAG_T_READWRITE;
81954|             }
81955|
      |
81956|             // see if registry has an override.
81957|             ULONG Temp;
81958|

```



```

    | Reg_GetULONGKey(gRegistryPath,L"SnapShotFlags",PsmIn->Sn
    | apShotFlags,&Temp);
81959|         PsmIn->SnapShotFlags = UCHAR(Temp &
    | 0xff);
81960|
81961|         // not used fields
81962|         wcscpy(PsmIn->CacheFileName,L "");
81963|         PsmIn->SizeOfCacheFileMB=0;
81964|         PsmIn->MaxSizeOfCacheFileMB=0;
81965|         PsmIn->QuiescentWait = 0;
81966|         PsmIn->QuiescentTimeout=0;
81967|
81968|         // set snapshot properties
81969|         PsmIn->DllPrivateUse=0;
81970|
    | PsmIn->CallerPrivateUse=(void*)'BmtO'; // set the group
81971|         PsmIn->NumToKeep=-1;
81972|         PsmIn->Priority=0;
81973|         PsmIn->Reserved1=0;
81974|         PsmIn->Reserved2=0;
81975|
81976|         PsmIn->NumberOfDevices =
    | In->NumberOfDevices;
81977|
81978|         WCHAR
    | *p=(WCHAR*)(PsmIn->DeviceName+In->NumberOfDevices);
81979|         for(ULONG
    | i=0;i<In->NumberOfDevices;i++) {
81980|
    | wcscpy(p,(WCHAR*)DN_MakePointer(In,In->DeviceName[i]));
81981|         PsmIn->DeviceName[i] =
    | DN_MakeOffset(PsmIn,p);
81982|         p+=wcslen(p)+1;
81983|     }
81984|
81985|         newIrp =
    | IoBuildDeviceIoControlRequest (
81986|             IOCTL_TURNON_PSM,
81987|             PsmDeviceObject,
81988|             PsmIn,
81989|             65536,
81990|             PsmIn,
81991|             65536,
81992|             FALSE,
81993|             OtmToPsm->Event,
81994|             &OtmToPsm->IoStatus );
81995|
81996|         if(newIrp) {
81997|
    | Debug(DEBUG_OTM_FILTER,("Bridge: CreateSnapShot:

```

```

    | OtmIrp=%08x, PsmIrp=%08x,
    | OtmToPsm=%08x\n",Irp,newIrp,OtmToPsm));
81998|
81999|         HANDLE TempHandle;
82000|
82001|         Status = PsCreateSystemThread(
82002|
    | &TempHandle,THREAD_ALL_ACCESS,NULL,NULL,NULL,
82003|
    | (PKSTART_ROUTINE)OtmToPsmCreateSnapShotThread, // IN
    | PKSTART_ROUTINE StartRoutine,
82004|         (PVOID)OtmToPsm
    | // IN PVOID StartContext
82005|         );
82006|
82007|         if(NT_SUCCESS(Status)) {
82008|             // we dont care about the
    | thread status
82009|             ZwClose(TempHandle);
82010|
82011|             // its pending so do not
    | complete the request
82012|             Irp->IoStatus.Status =
    | STATUS_PENDING;
82013|             IoMarkIrpPending(Irp);
82014|
82015|             Status = IoCallDriver
    | (PsmDeviceObject, newIrp);
82016|
    | Debug(DEBUG_OTM_FILTER,"Bridge: Psm Create returned
    | %08x\n",Status));
82017|             return STATUS_PENDING;
82018|         } else {
82019|
    | Debug(DEBUG_OTM_FILTER,"Bridge: CreateSnapShot: Error
    | %08x starting thread\n",Status));
82020|             IoFreeIrp(newIrp);
82021|         }
82022|     } else {
82023|         Status =
    | STATUS_INSUFFICIENT_RESOURCES;
82024|     }
82025|     ExFreePool(OtmToPsm->Event);
82026| } else {
82027|     Status =
    | STATUS_INSUFFICIENT_RESOURCES;
82028| }
82029| ExFreePool(OtmToPsm);
82030| } else {
82031|     Status = STATUS_INSUFFICIENT_RESOURCES;

```

```

82032|     }
82033| }
82034| } __except( EXCEPTION_EXECUTE_HANDLER ) {
82035|     Status = GetExceptionCode();
82036|     Debug(DEBUG_OTM_FILTER,("Bridge: OtmTurnOnOtm:
    | Error! Exception %08x\n", Status));
82037| }
82038| #else
82039|     NTSTATUS Status=STATUS_INVALID_PARAMETER;
82040| #endif
82041|
82042|     Irp->IoStatus.Information = 0;
82043|     Irp->IoStatus.Status = Status;
82044|     IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
82045|     return Status;
82046| }
82047| /*-----
    | -----*/
82048| NTSTATUS OtmTurnOffOtm(PIRP Irp)
82049| {
82050|     NTSTATUS Status = STATUS_SUCCESS;
82051|     __try {
82052|         tClosePSMInternal *Buffer=NULL;
82053|         PIO_STACK_LOCATION plrpStack =
            | IoGetCurrentIrpStackLocation (Irp) ;
82054|
82055|         Debug(DEBUG_PROCCALL |
            | DEBUG_OTM_FILTER,("Bridge: TurnOffOtm\n"));
82056|         // METHOD_BUFFERED
82057|         // Irp->AssociatedIrp.SystemBuffer =
            | Input/Output buffer
82058|
82059|         if
            | (plrpStack->Parameters.DeviceIoControl.InputBufferLength
            | < sizeof(tCloseOTMInternal)) {
82060|             Status = PsmDestroySnapShot(NULL);
82061|         } else {
82062|             Buffer = (tClosePSMInternal
            | *)Irp->AssociatedIrp.SystemBuffer;
82063|             Status =
            | PsmDestroySnapShot(Buffer->KernelSnapShotPointer);
82064|         }
82065|     } __except( EXCEPTION_EXECUTE_HANDLER ) {
82066|         Status = GetExceptionCode();
82067|         Debug(DEBUG_OTM_FILTER,("Bridge: OtmTurnOffOtm:
            | Error! Exception %08x\n", Status));
82068|     }
82069|
82070|     Irp->IoStatus.Information = 0;
82071|     Irp->IoStatus.Status = Status;

```

```

82072|   IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
82073|   return Status;
82074|
82075| }
82076| /*-----*/
82077| | -----*/
82077| NTSTATUS OtmGetError(PIRP Irp)
82078| {
82079|   NTSTATUS Status = STATUS_SUCCESS;
82080|   ULONG RetCode=0;
82081|
82082|   __try {
82083|       PIO_STACK_LOCATION plrpStack =
82084|       | IoGetCurrentIrpStackLocation (Irp) ;
82084|       tOTM_GetErrorOut *Buffer=NULL;
82085|       PVOID Master=NULL;
82086|       // METHOD_OUT_DIRECT
82087|       // Irp->MdlAddress = Output buffer
82088|       // Irp->AssociatedIrp.SystemBuffer = Input
82089|       | buffer
82089|
82090|       if
82091|       | (plrpStack->Parameters.DeviceIoControl.InputBufferLength
82092|       | >= sizeof(PVOID)) {
82091|           Master =
82092|           | *((PVOID*)(Irp->AssociatedIrp.SystemBuffer));
82092|       }
82093|
82094|       Debug(DEBUG_PROCCALL |
82095|       | DEBUG_OTM_FILTER,("Bridge: GetError %08x\n",Master));
82095|
82096|       if
82097|       | (plrpStack->Parameters.DeviceIoControl.OutputBufferLengt
82098|       | h < sizeof(tOTM_GetErrorOut)) {
82097|           Debug(DEBUG_OTM_FILTER,("Bridge: Error!
82098|           | Output IOCTL buffer not big enough\n"));
82098|           Status = STATUS_INVALID_BUFFER_SIZE;
82099|       } else {
82100|           /*lint -save -e641 */
82101|           Buffer = (tOTM_GetErrorOut
82102|           | *)MmGetSystemAddressForMdlSafe( Irp->MdlAddress,
82103|           | NormalPagePriority );
82102|           /*lint -restore */
82103|
82104|           if (Buffer) {
82105|               Status =
82106|               | PsmGetError(Master,(pPSM_GetErrorOut)Buffer);
82106|
82107|               RetCode = sizeof(tOTM_GetErrorOut);
82108|           } else {

```

```

82109|         Debug(DEBUG_OTM_FILTER,("Bridge: Error!
| Buffer is NULL\n"));
82110|         Status = STATUS_INVALID_PARAMETER;
82111|     }
82112| }
82113|
82114| } __except( EXCEPTION_EXECUTE_HANDLER ) {
82115|     Status = GetExceptionCode();
82116|     Debug(DEBUG_OTM_FILTER,("Bridge: OtmGetError:
| Error! Exception %08x\n", Status));
82117| }
82118| // same here, except it will get converted to the
| win32 error code
82119| // by the io manager for us. this is the reason we
| use METHOD_OUT_DIRECT
82120| // is so we can return a error code, otherwise,
| when buffering, the buffer
82121| // will not be copied into the user buffer.
82122| Irp->IoStatus.Status = Status;
82123| Irp->IoStatus.Information = RetCode;
82124| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
82125| return Status;
82126| }
82127|
82128| /*-----
| -----*/
82129| NTSTATUS OtmSetWin32Link(PIRP Irp)
82130| {
82131|     NTSTATUS Status=STATUS_INVALID_BUFFER_SIZE;
82132|
82133|     __try {
82134|         PIO_STACK_LOCATION pIrpStack =
| IoGetCurrentIrpStackLocation (Irp) ;
82135|         pOTM_SetWin32Link
| Link=(pOTM_SetWin32Link)Irp->AssociatedIrp.SystemBuffer;
82136|         HANDLE TempHandle;
82137|
82138|         | ASSERT(sizeof(tOTM_SetWin32Link)==sizeof(tPSM_SetWin32Li
| nk));
82139|         if
| (pIrpStack->Parameters.DeviceIoControl.InputBufferLength
| >= sizeof(tOTM_SetWin32Link)) {
82140|             Status =
| PsmSetWin32Link((pPSM_SetWin32Link)Link);
82141|         }
82142|     } __except( EXCEPTION_EXECUTE_HANDLER ) {
82143|         Status = GetExceptionCode();
82144|         Debug(DEBUG_OTM_FILTER,("Bridge:
| OtmSetWin32Link: Error! Exception %08x\n", Status));

```

```

82145| }
82146|
82147| Irp->IoStatus.Information = 0;
82148| Irp->IoStatus.Status = Status;
82149| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
82150| return Status;
82151| }
82152| /*-----
| -----*/
82153| NTSTATUS OtmGetProgress(PIRP Irp)
82154| {
82155|     NTSTATUS Status = STATUS_SUCCESS;
82156|     ULONG RetSize=0;
82157|
82158|     __try {
82159|         PIO_STACK_LOCATION plrpStack =
| IoGetCurrentIrpStackLocation (Irp) ;
82160|         tPSM_GetProgressOut PsmProg;
82161|         tOTM_GetProgressOut *OtmProg;
82162|         PVOID Master=NULL;
82163|
82164|         // METHOD_BUFFERED
82165|         // Irp->AssociatedIrp.SystemBuffer =
| Input/Output buffer
82166|
82167|         if
| (plrpStack->Parameters.DeviceIoControl.InputBufferLength
| >= sizeof(PVOID)) {
82168|             Master =
| *((PVOID*)(Irp->AssociatedIrp.SystemBuffer));
82169|         }
82170|
82171|         Debug(DEBUG_PROCCALL |
| DEBUG_OTM_FILTER,("Bridge: OtmGetProgress
| %08x\n",Master));
82172|
82173|         if
| (plrpStack->Parameters.DeviceIoControl.OutputBufferLengt
| h < sizeof(tOTM_GetProgressOut)) {
82174|             Debug(DEBUG_OTM_FILTER,("Bridge: Error!
| IOCTL buffer not big enough\n"));
82175|             Status = STATUS_INVALID_BUFFER_SIZE;
82176|         } else {
82177|             OtmProg = (tOTM_GetProgressOut
| *)Irp->AssociatedIrp.SystemBuffer;
82178|             ASSERT(OtmProg);
82179|
82180|             Status = PsmGetProgress( Master, &PsmProg
| );
82181|             if(NT_SUCCESS(Status)) {

```

```

82182|         OtmProg->OnSecond          =
      | PsmProg.OnSecond;
82183|         OtmProg->OutOfSeconds        =
      | PsmProg.OutOfSeconds;
82184|         OtmProg->CacheFileSize       =
      | PsmProg.CacheFileSize;
82185|         OtmProg->CurrentCacheFileSize =
      | PsmProg.CurrentCacheFileSize;
82186|         RetSize = sizeof(*OtmProg);
82187|     }
82188| }
82189| } __except( EXCEPTION_EXECUTE_HANDLER ) {
82190|     Status = GetExceptionCode();
82191|     Debug(DEBUG_OTM_FILTER,("Bridge:
      | OtmGetProgress: Error! Exception %08x\n", Status));
82192| }
82193|
82194|
82195| Irp->IoStatus.Information = RetSize;
82196| Irp->IoStatus.Status = Status;
82197| IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
82198| return Status;
82199| }
82200|
82201| /*-----
      | -----*/
82202| NTSTATUS BridgeDeviceControl(PDEVICE_OBJECT
      | DeviceObject,PIRP Irp)
82203| {
82204|     NTSTATUS Status=STATUS_INVALID_DEVICE_REQUEST;
82205|     PIO_STACK_LOCATION pIrpStack =
      | IoGetCurrentIrpStackLocation (Irp) ;
82206|
82207|     Debug(DEBUG_PROCCALL | DEBUG_OTM_FILTER,("Bridge:
      | BridgeDeviceControl Called Dev=%p,
      | Irp=%p\n",DeviceObject,Irp));
82208|
82209|     | switch(pIrpStack->Parameters.DeviceIoControl.IoControlCo
      | de) {
82210|         case OTM_IOCTL_GET_VERSION    : Status =
      | OtmGetVersion(Irp); break;
82211|         case OTM_IOCTL_TURNON_OTM     : Status =
      | OtmTurnOnOtm(Irp); break;
82212|         case OTM_IOCTL_TURNOFF_OTM    : Status =
      | OtmTurnOffOtm(Irp); break;
82213|         case OTM_IOCTL_GET_ERROR      : Status =
      | OtmGetError(Irp); break;
82214|         case OTM_IOCTL_SET_WIN32_LINK : Status =
      | OtmSetWin32Link(Irp); break;

```

```

82215|     case OTM_IOCTL_GET_PROGRESS : Status =
      | OtmGetProgress(Irp); break;
82216|     case OTM_IOCTL_GET_VOLUME_STATS :
82217|     case OTM_IOCTL_GET_VOLUME_INFO :
82218|     case OTM_IOCTL_OPEN_EX      :
82219|     case OTM_IOCTL_CLOSE_EX     :
82220|     case OTM_IOCTL_FREE_VOLUME  :
82221|     case OTM_IOCTL_FREE_RANGES  :
82222|     case OTM_IOCTL_SET_FLUSH_ROUTINE:
82223|     case BRIDGE_IOCTL_IS_BRIDGE_INSTALLED:
82224|         Status = STATUS_SUCCESS;
82225|         Irp->IoStatus.Information = 0;
82226|         Irp->IoStatus.Status = Status;
82227|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
82228|         break;
82229|     default:
82230|         Irp->IoStatus.Information = 0;
82231|         Irp->IoStatus.Status = Status;
82232|         IoCompleteRequest (Irp, IO_NO_INCREMENT) ;
82233|     }
82234|
82235|     return Status;
82236| }
82237|
82238| /*--- end of file otm.cpp ---*/
82239|
82240|
82241|
82242| File Listing: otm.h
82243|
82244| NTSTATUS OtmGetVersion( PIRP Irp );
82245| NTSTATUS OtmTurnOnOtm( PIRP Irp );
82246| NTSTATUS OtmTurnOffOtm( PIRP Irp );
82247| NTSTATUS OtmGetError( PIRP Irp );
82248| NTSTATUS OtmSetWin32Link( PIRP Irp );
82249| NTSTATUS OtmGetProgress( PIRP Irp );
82250|
82251| /*--- end of file otm.h ---*/
82252|
82253|
82254|
82255| File Listing: otm_private.h
82256|
82257| #include <devioctl.h>
82258|
82259| /*lint -save -e43*/
82260| // otm ioctls
82261| #define OTM_IOCTL_GET_VERSION
      | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA0,METHOD_BUFFERED,FILE_
      | ANY_ACCESS)

```



```

82262| #define OTM_IOCTL_TURNON_OTM
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA1,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82263| #define OTM_IOCTL_TURNOFF_OTM
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA2,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82264| #define OTM_IOCTL_GET_ERROR
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA3,METHOD_OUT_DIRECT,FIL
    | E_ANY_ACCESS)
82265| //#define OTM_IOCTL_GET_STATS
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA4,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82266| #define OTM_IOCTL_SET_WIN32_LINK
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA6,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82267| #define OTM_IOCTL_OPEN_EX
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA7,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82268| #define OTM_IOCTL_CLOSE_EX
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA8,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82269| #define OTM_IOCTL_FREE_VOLUME
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA9,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82270| #define OTM_IOCTL_FREE_RANGES
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAB,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82271| #define OTM_IOCTL_GET_PROGRESS
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAC,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82272| //#define OTM_IOCTL_GET_OTM_INFO
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAD,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82273| #define OTM_IOCTL_SET_FLUSH_ROUTINE
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAE,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82274| #define OTM_IOCTL_GET_VOLUME_STATS
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAF,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82275| #define OTM_IOCTL_GET_VOLUME_INFO
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB0,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82276|
82277| #define OTM_IOCTL_BUG_CHECK
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xFFE,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82278| #if MEMDBG
82279| #define OTM_IOCTL_GET_MEMORY_USAGE
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xFFF,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)

```

```

82280| #endif
82281|
82282| // vdisk ioctls
82283| #define OTM_IOCTL_UNWRITE_PROTECT
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB6,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82284| #define OTM_IOCTL_WRITE_PROTECT
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB7,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82285| #define OTM_IOCTL_IS_OTM_VOLUME
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB8,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82286|
82287|
82288| // OTM bridge specific ioctls
82289| #define BRIDGE_IOCTL_IS_BRIDGE_INSTALLED
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0x900,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82290|
82291|
82292| #define OTM_IFLAG_NEW_SNAPSHOT 1
82293|
82294| // if this number goes above 15 then we need to change
    | OtManBitMapUseCount
82295| #define OTM_MAX_NUMBER_OF_SNAPSHOTS 15
82296|
82297| #define OTM_LINK_SetLink 0
82298| #define OTM_LINK_DeleteLink 1
82299|
82300| typedef struct sOTM_SetWin32Link {
82301|     ULONG Operation;
82302|     WCHAR Win32Link[256];
82303|     WCHAR NTDeviceName[256];
82304| } tOTM_SetWin32Link,*pOTM_SetWin32Link;
82305|
82306| typedef struct sOTM_FreeVolume {
82307|     PVOID KernelSnapShotPointer;
82308|     WCHAR VolumeName[256];
82309| } tOTM_FreeVolume,*pOTM_FreeVolume;
82310|
82311|
82312| #pragma warning (push)
82313| #pragma warning (disable:4200)
82314| /*lint -save -e43 -e1501 */
82315| typedef struct sOTMOpenTransactionInInternal {
82316|     DWORD Size;
82317|     WCHAR CacheFileName[256];
82318|     DWORD SizeOfCacheFileMB;
82319|     DWORD MaxSizeOfCacheFileMB;
82320|     DWORD Flags;

```

```

82321|   DWORD           InternalFlags;
82322|   DWORD           QuiescentWait;
82323|   DWORD           QuiescentTimeout;
82324|   HANDLE          ErrorEvent;
82325|   HANDLE          AbortEvent;
82326|   ULONG           NumberOfDevices;
82327|   ULONG           DeviceName[]; // offset from
      | beginning of this structure
82328| } tOTMOpenTransactionInInternal,
      | *pOTMOpenTransactionInInternal;
82329|
82330| typedef struct sOTMOpenTransactionOutInternal {
82331|   PVOID KernelSnapShotPointer; // cant access kernel
      | mode addresses in user mode
82332|   LARGE_INTEGER   SnapShotTime; // time snapshot
      | occurred.
82333|   ULONG           Instance;      // instance number
      | for this snapshot for volume mapping
82334|   WCHAR           CacheFileName[256]; // cache file
      | name being used
82335|   ULONG           NumberOfDevices;
82336|   ULONG           DeviceName[]; // offset from
      | beginning of this structure
82337| } tOTMOpenTransactionOutInternal,
      | *pOTMOpenTransactionOutInternal;
82338| /*lint -restore*/
82339| #pragma warning (pop)
82340|
82341| typedef struct sCloseOTMInternal {
82342|   PVOID KernelSnapShotPointer; // cant access kernel
      | mode addresses in user mode
82343| } tCloseOTMInternal, *pCloseOTMInternal;
82344|
82345| typedef struct sOTMVolumelInfoIn {
82346|   unsigned short Size;
82347|   unsigned short Version;
82348|   WCHAR VolumeName[256];
82349| } tOTMVolumelInfoIn, *pOTMVolumelInfoIn;
82350|
82351|
82352| typedef struct sOTMVolumelInfoOut {
82353|   ULONG SectorsPerCluster;
82354|   ULONG Cluster0Offset;
82355| } tOTMVolumelInfoOut, *pOTMVolumelInfoOut;
82356|
82357| #if MEMDBG
82358| typedef struct sOTMGetMemoryUsageOut {
82359|   ULONG MemTotalNonpagedAlloced;
82360|   ULONG MemTotalPagedAlloced;
82361|   ULONG MemMaxNonpagedAlloced;

```

```

82362|    ULONG MemMaxPagedAlloced;
82363| } tOTMGetMemoryUsageOut, *pOTMGetMemoryUsageOut;
82364|
82365| #endif
82366|
82367| typedef struct sOTMSetFlushRoutine {
82368|     NTSTATUS (*ZwFlushBuffersFile)( IN HANDLE
82369|         | FileHandle, OUT PIO_STATUS_BLOCK IoStatusBlock );
82370| } tOTMSetFlushRoutine, *pOTMSetFlushRoutine;
82371|
82372| typedef struct sOTM_GetErrorOut {
82373|     DWORD      ErrorCode;
82374| } tOTM_GetErrorOut, *pOTM_GetErrorOut;
82375|
82376| typedef struct sOTM_GetProgressOut {
82377|     DWORD      OnSecond;
82378|     DWORD      OutOfSeconds;
82379|     DWORD      CacheFileSize;
82380|     DWORD      CurrentCacheFileSize;
82381| } tOTM_GetProgressOut, *pOTM_GetProgressOut;
82382|
82383| typedef struct _sOTM_VersionInfo {
82384|     DWORD      Size;
82385|     DWORD      LoVersion;
82386|     DWORD      Version;
82387|     DWORD      OSType;
82388|     DWORD      OSVersion;
82389|     DWORD      CommunicationMethods;
82390| } tOTM_VersionInfo, *pOTM_VersionInfo;
82391|
82392| #define OTM_SUPPORTED_VERSION_NUMBER    0x0111
82393| #define OTM_SUPPORTED_LOW_VERSION_NUMBER 0x0110
82394|
82395|
82396| /*lint -restore*/
82397|
82398| /*--- end of file otm_private.h ---*/
82399|
82400|
82401|
82402| File Listing: otmbridge.h
82403|
82404| //File Name: messages.mc
82405| //
82406| // Note: comments in the .mc file must use both ";"
82407| //      | and "/*".
82408| //
82409| // Status values are 32 bit values layed out as
82410| // follows:

```

```

82409| //
82410| //  3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
82411| //  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7
      | 6 5 4 3 2 1 0
82412| //
      | +---+ +-----+ +-----+
      | -----+
82413| // |Sev|C|      Facility      |
      | Code      |
82414| //
      | +---+ +-----+ +-----+
      | -----+
82415| //
82416| // where
82417| //
82418| //   Sev - is the severity code
82419| //
82420| //       00 - Success
82421| //       01 - Informational
82422| //       10 - Warning
82423| //       11 - Error
82424| //
82425| //   C - is the Customer code flag
82426| //
82427| //   Facility - is the facility code
82428| //
82429| //   Code - is the facility's status code
82430| //
82431|
82432|
82433| //
82434| // %1 is reserved by the IO Manager. If
      | IoAllocateErrorLogEntry is
82435| // called with a device, the name of the device will be
      | inserted into
82436| // the message at %1. Otherwise, the place of %1 will
      | be left empty.
82437| // In either case, the insertion strings from the
      | driver's error log
82438| // entry starts at %2. In other words, the first
      | insertion string goes
82439| // to %2, the second to %3 and so on.
82440| //
82441|
82442| //
82443| // Values are 32 bit values layed out as follows:
82444| //
82445| //  3 3 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1
82446| //  1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7
      | 6 5 4 3 2 1 0

```

```

82447| //
      | +---+---+-----+-----+
      | -----+
82448| // |Sev|C|R|   Facility       |
      | Code       |
82449| //
      | +---+---+-----+-----+
      | -----+
82450| //
82451| // where
82452| //
82453| //   Sev - is the severity code
82454| //
82455| //       00 - Success
82456| //       01 - Informational
82457| //       10 - Warning
82458| //       11 - Error
82459| //
82460| //   C - is the Customer code flag
82461| //
82462| //   R - is a reserved bit
82463| //
82464| //   Facility - is the facility code
82465| //
82466| //   Code - is the facility's status code
82467| //
82468| //
82469| // Define the facility codes
82470| //
82471| #define FACILITY_RPC_STUBS           0x3
82472| #define FACILITY_RPC_RUNTIME         0x2
82473| #define FACILITY_IO_ERROR_CODE       0x4
82474|
82475|
82476| //
82477| // Define the severity codes
82478| //
82479| #define STATUS_SEVERITY_WARNING        0x2
82480| #define STATUS_SEVERITY_SUCCESS        0x0
82481| #define STATUS_SEVERITY_INFORMATIONAL  0x1
82482| #define STATUS_SEVERITY_ERROR          0x3
82483|
82484|
82485| //
82486| // MessageId: BRIDGE_SUCCESS
82487| //
82488| // MessageText:
82489| //
82490| // %2
82491| //

```

```
82492| #define BRIDGE_SUCCESS
      | ((NTSTATUS)0x20000001L)
82493|
82494| //
82495| // MessageId: BRIDGE_INFO
82496| //
82497| // MessageText:
82498| //
82499| // %2
82500| //
82501| #define BRIDGE_INFO
      | ((NTSTATUS)0x60000001L)
82502|
82503| //
82504| // MessageId: BRIDGE_ACTIVATED
82505| //
82506| // MessageText:
82507| //
82508| // OTM to PSM Bridge installed and activated.
82509| //
82510| #define BRIDGE_ACTIVATED
      | ((NTSTATUS)0x60000002L)
82511|
82512| //
82513| // MessageId: BRIDGE_DEACTIVATED
82514| //
82515| // MessageText:
82516| //
82517| // OTM to PSM Bridge deactivated.
82518| //
82519| #define BRIDGE_DEACTIVATED
      | ((NTSTATUS)0x60000003L)
82520|
82521| //
82522| // MessageId: BRIDGE_WARNING
82523| //
82524| // MessageText:
82525| //
82526| // %2
82527| //
82528| #define BRIDGE_WARNING
      | ((NTSTATUS)0xA0000001L)
82529|
82530| //
82531| // MessageId: BRIDGE_ERROR
82532| //
82533| // MessageText:
82534| //
82535| // %2
82536| //
```

```
82537| #define BRIDGE_ERROR
      | ((NTSTATUS)0xE0000001L)
82538|
82539| //
82540| // MessageId: BRIDGE_WRONG_PSM_VERSION
82541| //
82542| // MessageText:
82543| //
82544| // An incompatible version of PSM is installed
82545| //
82546| #define BRIDGE_WRONG_PSM_VERSION
      | ((NTSTATUS)0xE0000002L)
82547|
82548| //
82549| // MessageId: BRIDGE_WRONG_OTM_VERSION
82550| //
82551| // MessageText:
82552| //
82553| // An incompatible version of OTM is installed
82554| //
82555| #define BRIDGE_WRONG_OTM_VERSION
      | ((NTSTATUS)0xE0000003L)
82556|
82557| /*--- end of file otmbridge.h ---*/
82558|
82559|
82560|
82561| File Listing: precomp.h
82562|
82563| extern "C" {
82564|     #include <ddk\ntddk.h>
82565|     #include <stdarg.h>
82566|     #include <stdio.h>
82567| }
82568|
82569| #define DWORD ULONG
82570| #define BYTE UCHAR
82571|
82572| #include "init.h"
82573| #include "otmbridge.h"
82574| #include "support.h"
82575| #include "otm.h"
82576| #include "psm.h"
82577|
82578| /*--- end of file precomp.h ---*/
82579|
82580|
82581|
82582| File Listing: psm.cpp
82583|
```



```

82584| #include <precomp.h>
82585|
82586| NTSTATUS PsmDestroySnapShot( PVOID SnapShot )
82587| {
82588|     PIRP                Irp=NULL;
82589|     KEVENT               Event={0};
82590|     IO_STATUS_BLOCK       IoStatusBlock={0};
82591|     NTSTATUS              Status=0;
82592|     tClosePSMInternal     Close;
82593|
82594|     __try {
82595|
82596|         Close.KernelSnapShotPointer = SnapShot;
82597|
82598|         //
82599|         // Set the event object to the unsigned
82600|         | state.
82601|         // It will be used to signal request
82602|         | completion.
82603|         //
82604|         KeInitializeEvent(&Event,
82605|                           NotificationEvent,
82606|                           FALSE);
82607|
82608|
82609|         //
82610|         // Create IRP for get drive layout device
82611|         | control.
82612|         //
82613|         Irp =
82614|         | IoBuildDeviceIoControlRequest(IOCTL_TURNOFF_PSM,
82615|         | PsmDeviceObject,
82616|         | &Close,
82617|         | sizeof(Close),
82618|         | NULL,
82619|         | 0,
82620|         | FALSE,
82621|         | &Event,
82622|         | &IoStatusBlock);
82623|         if (!Irp) {
82624|             Debug(DEBUG_DEVSUP,("PsmDestroySnapShot:
82625|             | Error! Unable to allocate irp\n"));
82626|             return STATUS_INSUFFICIENT_RESOURCES;

```

```

82626|     }
82627|
82628|     Status = IoCallDriver(PsmDeviceObject, Irp);
82629|
82630|     if (Status == STATUS_PENDING) {
82631|
82632|         ASSERT(KeGetCurrentIrql() <
82633|             | DISPATCH_LEVEL);
82634|         pmWaitForSingleObject(&Event,NULL);
82635|
82636|         Status = IoStatusBlock.Status;
82637|     }
82638| } __except(EXCEPTION_EXECUTE_HANDLER) {
82639|     Status = GetExceptionCode();
82640|     Debug(DEBUG_DEVSUP,("PsmDestroySnapShot:
82641|         | Exception %08x\n",Status));
82642| }
82643| return Status;
82644| }
82645| NTSTATUS PsmGetError( PVOID SnapShot, tPSM_GetErrorOut
82646|     | *Error )
82647| {
82648|     PIRP                Irp=NULL;
82649|     KEVENT               Event={0};
82650|     IO_STATUS_BLOCK      IoStatusBlock={0};
82651|     NTSTATUS              Status=0;
82652|     tClosePSMInternal     Close;
82653|
82654|     __try {
82655|
82656|         Close.KernelSnapShotPointer = SnapShot;
82657|
82658|         //
82659|         // Set the event object to the unsignaled
82660|         | state.
82661|         // It will be used to signal request
82662|         | completion.
82663|         //
82664|         KeInitializeEvent(&Event,
82665|             NotificationEvent,
82666|             FALSE);
82667|
82668|         //
82669|         // Create IRP for get drive layout device
82670|         | control.
82671|         //

```

```

82670|
82671|     Irp =
82672|         | IoBuildDeviceIoControlRequest(IOCTL_GET_ERROR,
82673|         | PsmDeviceObject,
82674|         | &Close,
82675|         | sizeof(Close),
82676|         | Error,
82677|         | sizeof(*Error),
82678|         | FALSE,
82679|         | &Event,
82680|         | &IoStatusBlock);
82681|     if (!Irp) {
82682|         Debug(DEBUG_DEVSUP,("PsmGetError: Error!
82683|         | Unable to allocate irp\n"));
82684|         return STATUS_INSUFFICIENT_RESOURCES;
82685|     }
82686|     Status = IoCallDriver(PsmDeviceObject, Irp);
82687|     if (Status == STATUS_PENDING) {
82688|         ASSERT(KeGetCurrentIrql() <
82689|         | DISPATCH_LEVEL);
82690|         pmWaitForSingleObject(&Event,NULL);
82691|         Status = IoStatusBlock.Status;
82692|     }
82693| } __except(EXCEPTION_EXECUTE_HANDLER) {
82694|     Status = GetExceptionCode();
82695|     Debug(DEBUG_DEVSUP,("PsmGetError: Exception
82696|     | %08x\n",Status));
82697| }
82698| return Status;
82699| }
82700| }
82701|
82702| NTSTATUS PsmSetWin32Link( tPSM_SetWin32Link *Link )
82703| {
82704|     PIRP             Irp=NULL;
82705|     KEVENT            Event={0};
82706|     IO_STATUS_BLOCK   IoStatusBlock={0};
82707|     NTSTATUS          Status=0;
82708|
82709|     __try {
82710|
82711|         //

```

```

82712|    // Set the event object to the unsigaled
      | state.
82713|    // It will be used to signal request
      | completion.
82714|    //
82715|
82716|    KeInitializeEvent(&Event,
82717|                     NotificationEvent,
82718|                     FALSE);
82719|
82720|
82721|
82722|    //
82723|    // Create IRP for get drive layout device
      | control.
82724|    //
82725|
82726|    Irp =
      | IoBuildDeviceIoControlRequest(IOCTL_SET_WIN32_LINK,
82727|    | PsmDeviceObject,
82728|                                     Link,
82729|
      | sizeof(*Link),
82730|                                     NULL,
82731|                                     0,
82732|                                     FALSE,
82733|                                     &Event,
82734|
      | &IoStatusBlock);
82735|
82736|    if (!Irp) {
82737|        Debug(DEBUG_DEVSUP,("PsmSetWin32Link:
      | Error! Unable to allocate irp\n"));
82738|        return STATUS_INSUFFICIENT_RESOURCES;
82739|    }
82740|
82741|    Status = IoCallDriver(PsmDeviceObject, Irp);
82742|
82743|    if (Status == STATUS_PENDING) {
82744|
82745|        ASSERT(KeGetCurrentIrql() <
      | DISPATCH_LEVEL);
82746|        pmWaitForSingleObject(&Event, NULL);
82747|
82748|        Status = IoStatusBlock.Status;
82749|    }
82750| } __except(EXCEPTION_EXECUTE_HANDLER) {
82751|     Status = GetExceptionCode();
82752|     Debug(DEBUG_DEVSUP,("PsmSetWin32Link: Exception

```

```

    | %08x\n",Status));
82753|    }
82754|    return Status;
82755| }
82756|
82757| NTSTATUS PsmGetProgress( PVOID SnapShot,
    | tPSM_GetProgressOut *Progress )
82758| {
82759|     PIRP                Irp=NULL;
82760|     KEVENT               Event={0};
82761|     IO_STATUS_BLOCK       IoStatusBlock={0};
82762|     NTSTATUS              Status=0;
82763|     tClosePSMInternal     Close;
82764|
82765|     __try {
82766|
82767|         Close.KernelSnapShotPointer = SnapShot;
82768|
82769|         //
82770|         // Set the event object to the unsigned
    | state.
82771|         // It will be used to signal request
    | completion.
82772|         //
82773|
82774|         KeInitializeEvent(&Event,
82775|             NotificationEvent,
82776|             FALSE);
82777|
82778|
82779|
82780|         //
82781|         // Create IRP for get drive layout device
    | control.
82782|         //
82783|
82784|         Irp =
    | IoBuildDeviceIoControlRequest(IOCTL_GET_PROGRESS,
82785|
    | PsmDeviceObject,
82786|
    | &Close,
82787|
    | sizeof(Close),
82788|
    | Progress,
82789|
    | sizeof(*Progress),
82790|
    | FALSE,
82791|
    | &Event,
82792|
    | &IoStatusBlock);

```

```

82793|
82794|     if (!Irp) {
82795|         Debug(DEBUG_DEVSUP,("PsmGetProgress: Error!
      | Unable to allocate irp\n"));
82796|         return STATUS_INSUFFICIENT_RESOURCES;
82797|     }
82798|
82799|     Status = IoCallDriver(PsmDeviceObject, Irp);
82800|
82801|     if (Status == STATUS_PENDING) {
82802|
82803|         ASSERT(KeGetCurrentIrql() <
      | DISPATCH_LEVEL);
82804|         pmWaitForSingleObject(&Event,NULL);
82805|
82806|         Status = IoStatusBlock.Status;
82807|     }
82808| } __except(EXCEPTION_EXECUTE_HANDLER) {
82809|     Status = GetExceptionCode();
82810|     Debug(DEBUG_DEVSUP,("PsmGetProgress: Exception
      | %08x\n",Status));
82811| }
82812| return Status;
82813| }
82814|
82815| NTSTATUS PsmGetVersion( tPSM_VersionInfo *Version )
82816| {
82817|     PIRP             Irp=NULL;
82818|     KEVENT            Event={0};
82819|     IO_STATUS_BLOCK   IoStatusBlock={0};
82820|     NTSTATUS          Status=0;
82821|
82822|     __try {
82823|
82824|         //
82825|         // Set the event object to the unsigned
      | state.
82826|         // It will be used to signal request
      | completion.
82827|         //
82828|
82829|         KeInitializeEvent(&Event,
82830|             NotificationEvent,
82831|             FALSE);
82832|
82833|         Version->Size = sizeof(*Version);
82834|
82835|         //
82836|         // Create IRP for get drive layout device
      | control.

```

```

82837|    //
82838|
82839|    Irp =
82840|        | IoBuildDeviceIoControlRequest(IOCTL_GET_VERSION,
82841|        | PsmDeviceObject,
82842|        | Version,
82843|        | sizeof(*Version),
82844|        | Version,
82845|        | sizeof(*Version),
82846|        | FALSE,
82847|        | &Event,
82848|        | &IoStatusBlock);
82849|    if (!Irp) {
82850|        Debug(DEBUG_DEVSUP,("PsmGetVersion: Error!
82851|        | Unable to allocate irp\n"));
82852|        return STATUS_INSUFFICIENT_RESOURCES;
82853|    }
82854|    Status = IoCallDriver(PsmDeviceObject, Irp);
82855|
82856|    if (Status == STATUS_PENDING) {
82857|
82858|        ASSERT(KeGetCurrentIrql() <
82859|        | DISPATCH_LEVEL);
82860|        pmWaitForSingleObject(&Event,NULL);
82861|        Status = IoStatusBlock.Status;
82862|    }
82863| } __except(EXCEPTION_EXECUTE_HANDLER) {
82864|     Status = GetExceptionCode();
82865|     Debug(DEBUG_DEVSUP,("PsmGetVersion: Exception
82866|     | %08x\n",Status));
82867| }
82868| return Status;
82869| }
82870| #ifdef NOT_USED
82871| PUNICODE_STRING UpperUnicodeString( PUNICODE_STRING Uni
82872| | )
82873| {
82874|     for(ULONG i=0;i<Uni->Length;i++) {
82875|         Uni->Buffer[0] =
82876|         | RtlUppcaseUnicodeChar(Uni->Buffer[0]);
82877|     }
82878|     return Uni;

```

```

82877| }
82878|
82879| BOOLEAN PsmlsVolumeExcluded( WCHAR *NTName )
82880| {
82881|     WCHAR Path[256];
82882|     UNICODE_STRING Result;
82883|     BOOLEAN Ret = FALSE;
82884|
82885|     | wcscpy(Path,L"\\Registry\\Machine\\System\\CurrentContro
      | ISet\\Services\\psman5\\persistent");
82886|
82887|     | Reg_GetStringKey(Path,L"ExcludedDrives",L"A:\\,B:\\",&Re
      | sult);
82888|
82889|     if(Result.Buffer) {
82890|         // FIXFIXFIX exclude list can also contain
      | volume guids,
82891|         // nt name, and volume mount points.
82892|
82893|         // incase there is no drive letter
82894|         wcscpy(Path,NTName);
82895|
82896|         GetDriveLetterFromNTName( NTName, Path,
      | sizeof(Path));
82897|
82898|         // the excluded list looks like
82899|         // a:\\,b:\\,c:\\
82900|         // returned Path looks like
82901|         // C:
82902|
82903|         wcscpy(Path,L"");
82904|
82905|         UpperUnicodeString(&Result);
82906|
82907|         if ( wcsstr(Result.Buffer,Path)!=NULL ) {
82908|             Ret = TRUE;
82909|         }
82910|
82911|         Reg_FreeString(&Result);
82912|     }
82913|
82914|     return Ret;
82915| }
82916|
82917| NTSTATUS PsmCreateFiles( tOpenTransactionInInternal *In
      | )
82918| {
82919|     PIRP                Irp=NULL;

```



```

82920|    KEVENT                Event={0};
82921|    IO_STATUS_BLOCK        IoStatusBlock={0};
82922|    NTSTATUS                Status=0;
82923|
82924|    __try {
82925|
82926|        //
82927|        // Set the event object to the unsigned
82928|        | state.
82929|        // It will be used to signal request
82930|        | completion.
82931|        //
82932|        KeInitializeEvent(&Event,
82933|                           NotificationEvent,
82934|                           FALSE);
82935|
82936|
82937|        //
82938|        // Create IRP for get drive layout device
82939|        | control.
82940|        //
82941|        Irp =
82942|        | IoBuildDeviceIoControlRequest(IOCTL_CREATE_FILES,
82943|        | PsmDeviceObject,
82944|        | In,
82945|        | sizeof(*In),
82946|        | NULL,
82947|        | 0,
82948|        | FALSE,
82949|        | &Event,
82950|        | &IoStatusBlock);
82951|        if (!Irp) {
82952|            Debug(DEBUG_DEVSUP,("PsmCreateFiles: Error!
82953|            | Unable to allocate irp\n"));
82954|            return STATUS_INSUFFICIENT_RESOURCES;
82955|        }
82956|        Status = IoCallDriver(PsmDeviceObject, Irp);
82957|
82958|        if (Status == STATUS_PENDING) {
82959|
82960|            ASSERT(KeGetCurrentIrql() <
82961|            | DISPATCH_LEVEL);

```

```

82961|         pmWaitForSingleObject(&Event,NULL);
82962|
82963|         Status = IoStatusBlock.Status;
82964|     }
82965| } __except(EXCEPTION_EXECUTE_HANDLER) {
82966|     Status = GetExceptionCode();
82967|     Debug(DEBUG_DEVSUP,("PsmCreateFiles: Exception
    | %08x\n",Status));
82968| }
82969| return Status;
82970| }
82971|
82972| #endif
82973|
82974| /*--- end of file psm.cpp ---*/
82975|
82976|
82977|
82978| File Listing: psm.h
82979|
82980| #define IOCTL_GET_VERSION
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA0,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82981| #define IOCTL_TURNON_PSM
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA1,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82982| #define IOCTL_TURNOFF_PSM
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA2,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82983| #define IOCTL_GET_ERROR
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA3,METHOD_OUT_DIRECT,FIL
    | E_ANY_ACCESS)
82984| #define IOCTL_SET_WIN32_LINK
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAA6,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82985| #define IOCTL_GET_PROGRESS
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAAC,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82986| #define IOCTL_CREATE_FILES
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAC7,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82987|
82988| // vdisk ioctls
82989| #define IOCTL_IS_PSM_VOLUME
    | CTL_CODE(FILE_DEVICE_UNKNOWN,0xAB8,METHOD_BUFFERED,FILE_
    | ANY_ACCESS)
82990|
82991| #define LINK_SetLink 0
82992| #define LINK_DeleteLink 1
82993|

```

```

82994| typedef struct sPSM_SetWin32Link {
82995|     ULONG Operation;
82996|     WCHAR Win32Link[256];
82997|     WCHAR NTDeviceName[256];
82998| } tPSM_SetWin32Link, *pPSM_SetWin32Link;
82999|
83000| #pragma warning (push)
83001| #pragma warning (disable:4200)
83002| typedef struct sOpenTransactionInInternal {
83003|     DWORD          Size;
83004|     WCHAR          CacheFileName[256];
83005|     DWORD          SizeOfCacheFileMB;
83006|     DWORD          MaxSizeOfCacheFileMB;
83007|     DWORD          Flags;
83008|     DWORD          InternalFlags;
83009|     DWORD          QuiescentWait;
83010|     DWORD          QuiescentTimeout;
83011|     HANDLE         ErrorEvent;
83012|     HANDLE         AbortEvent;
83013|     PVOID          DllPrivateUse;
83014|     PVOID          CallerPrivateUse;
83015|     ULONG          NumToKeep;
83016|     BYTE           Priority;
83017|     BYTE           SnapShotFlags;
83018|     BYTE           Reserved1;
83019|     BYTE           Reserved2; // padding
83020|     ULONG          NumberOfDevices;
83021|     ULONG          DeviceName[]; // offset from
        | beginning of this structure
83022| } tOpenTransactionInInternal,
        | *pOpenTransactionInInternal;
83023|
83024| typedef struct sOpenTransactionOutInternal {
83025|     PVOID KernelSnapShotPointer; // cant access kernel
        | mode addresses in user mode
83026|     LARGE_INTEGER SnapShotTime; // time snapshot
        | occurred.
83027|     ULONG          Instance; // instance number
        | for this snapshot for volume mapping
83028|     WCHAR          CacheFileName[256]; // cache file
        | name being used
83029|     ULONG          Persistent;
83030|     ULONG          NumberOfDevices;
83031|     ULONG          DeviceName[]; // offset from
        | beginning of this structure
83032| } tOpenTransactionOutInternal,
        | *pOpenTransactionOutInternal;
83033| /*lint -restore*/
83034| #pragma warning (pop)
83035|

```

```

83036| typedef struct sClosePSMInternal {
83037|     PVOID KernelSnapshotPointer; // cant access kernel
      | mode addresses in user mode
83038| } tClosePSMInternal, *pClosePSMInternal;
83039|
83040| typedef struct _sPSM_VersionInfo2 {
83041|     DWORD    Size;
83042|     DWORD    LoVersion;
83043|     DWORD    Version;
83044|     DWORD    OSType;
83045|     DWORD    OSVersion;
83046|     DWORD    CommunicationMethods;
83047|     DWORD    Eval:1;
83048|     DWORD    Expired:1;
83049|     DWORD    Reserved:30;
83050|     ULARGE_INTEGER ExpireDate;
83051| } tPSM_VersionInfo2, *pPSM_VersionInfo2;
83052|
83053| #define sPSM_VersionInfo sPSM_VersionInfo2
83054| #define tPSM_VersionInfo tPSM_VersionInfo2
83055| #define pPSM_VersionInfo pPSM_VersionInfo2
83056|
83057| typedef struct sPSM_GetErrorOut {
83058|     DWORD        ErrorCode;
83059| } tPSM_GetErrorOut, *pPSM_GetErrorOut;
83060|
83061| typedef struct sPSM_GetProgressOut {
83062|     DWORD    OnSecond;
83063|     DWORD    OutOfSeconds;
83064|     DWORD    CacheFileSize;
83065|     DWORD    CurrentCacheFileSize;
83066| } tPSM_GetProgressOut, *pPSM_GetProgressOut;
83067|
83068| #define PSM_IFLAG_NEW_SNAPSHOT    0x01
83069| #define PSM_IFLAG_PERSISTENT    0x02
83070|
83071| // version 2.0 snapshot flags
83072| #define PSM_2_00_SS_FLAG_READWRITE    0x00
83073| #define PSM_2_00_SS_FLAG_READONLY    0x01
83074| #define PSM_2_00_SS_FLAG_READWRITE_PERSISTENT    0x02
83075|
83076| // version 2.1+ snapshot flags
83077| #define PSM_SS_BIT_WRITEABLE    0x02
83078| #define PSM_SS_FLAG_T_READONLY    (0)
83079| #define PSM_SS_FLAG_T_READWRITE
      | (PSM_SS_BIT_WRITEABLE)
83080|
83081| #define DN_MakePointer(Start,Offset)
      | ((PVOID)(((char*)(Start))+(Offset)))
83082| #define DN_MakeOffset(Start,Offset)

```

```

| ((ULONG)(((char*)(Offset))-((char*)(Start))))
83083|
83084|
83085| NTSTATUS PsmDestroySnapShot( PVOID SnapShot );
83086| NTSTATUS PsmGetError( PVOID SnapShot, tPSM_GetErrorOut
| *Error );
83087| NTSTATUS PsmSetWin32Link( tPSM_SetWin32Link *Link );
83088| NTSTATUS PsmGetProgress( PVOID SnapShot,
| tPSM_GetProgressOut *Progress );
83089| NTSTATUS PsmGetVersion( tPSM_VersionInfo *Version );
83090| BOOLEAN PsmlsVolumeExcluded( WCHAR *NTName );
83091| NTSTATUS PsmCreateFiles( tOpenTransactionInInternal *In
| );
83092|
83093| /*--- end of file psm.h ---*/
83094|
83095|
83096|
83097| File Listing: resource.h
83098|
83099| //{NO_DEPENDENCIES}
83100| // Microsoft Developer Studio generated include file.
83101| // Used by SBPSMAN.RC
83102| //
83103| #define VER_PRODUCTBUILD 3
83104| #define VER_PRODUCTVERSION_W 0x0101
83105|
83106| // Next default values for new objects
83107| //
83108| #ifdef APSTUDIO_INVOKED
83109| #ifndef APSTUDIO_READONLY_SYMBOLS
83110| #define _APS_NO_MFC 1
83111| #define _APS_NEXT_RESOURCE_VALUE 101
83112| #define _APS_NEXT_COMMAND_VALUE 40001
83113| #define _APS_NEXT_CONTROL_VALUE 1000
83114| #define _APS_NEXT_SYMED_VALUE 101
83115| #endif
83116| #endif
83117|
83118|
83119|
83120| File Listing: support.cpp
83121|
83122| #include "precomp.h"
83123| // modified from psm\reg.cpp
83124| void Reg_GetULONGKey (
83125| IN PWCHAR RegistryPath,
83126| IN PWCHAR Key,
83127| IN ULONG Default,
83128| OUT PULONG Result

```

```

83129| )
83130| {
83131|     RTL_QUERY_REGISTRY_TABLE paramTable[2]={0};
83132|
83133|     RtlZeroMemory( &paramTable[0], sizeof(paramTable)
83134|         | );
83135|     paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT;
83136|     paramTable[0].Name = Key;
83137|     paramTable[0].EntryContext = Result;
83138|     paramTable[0].DefaultType = REG_DWORD;
83139|     paramTable[0].DefaultData = &Default;
83140|     paramTable[0].DefaultLength = sizeof(ULONG);
83141|
83142|     if (!NT_SUCCESS(RtlQueryRegistryValues(
83143|         RTL_REGISTRY_ABSOLUTE | RTL_REGISTRY_OPTIONAL,
83144|         RegistryPath,
83145|         &paramTable[0],
83146|         NULL,
83147|         NULL
83148|     ))) {
83149|         *Result = Default;
83150|     }
83151| }
83152|
83153| /*-----
83154| | -----*/
83155| // modified from psm\reg.cpp
83156| void Reg_GetStringKey (
83157|     IN PWCHAR RegistryPath,
83158|     IN PWCHAR Key,
83159|     IN PWCHAR Default,
83160|     OUT PUNICODE_STRING Result
83161| )
83162| /*++
83163| Routine Description:
83164|
83165|     Get a string from the registry
83166|
83167| Arguments:
83168|
83169|     RegistryPath  the path to query
83170|     Key           Key to query
83171|     Default       Default value to return if Key does
83172|                   | not exist
83173| Return Value:
83174|     The Key value
83175|

```

```

83176| --*/
83177|
83178| {
83179|     RTL_QUERY_REGISTRY_TABLE paramTable[2]={0};
83180|     NTSTATUS Status;
83181|
83182|     RtlZeroMemory( &paramTable[0], sizeof(paramTable)
83183|         | );
83184|     paramTable[0].Flags = RTL_QUERY_REGISTRY_DIRECT |
83185|         | RTL_QUERY_REGISTRY_NOEXPAND;
83186|     paramTable[0].Name = Key;
83187|     paramTable[0].EntryContext = Result;
83188|     paramTable[0].DefaultType = REG_SZ;
83189|     paramTable[0].DefaultData = Default;
83190|     paramTable[0].DefaultLength =
83191|         | wcslen(Default)*sizeof(WCHAR)+sizeof(WCHAR);
83192|
83193|     Result->Length = 0;
83194|     Result->MaximumLength=256;
83195|     Result->Buffer = (WCHAR
83196|         | *)ExAllocatePoolWithTag(PagedPool,256,'RmtO');
83197|
83198|     Debug(DEBUG_INFO,("Path='%S',
83199|         | Key='%S'\n",RegistryPath,Key));
83200|
83201|     if (!NT_SUCCESS(Status = RtlQueryRegistryValues(
83202|         | RTL_REGISTRY_ABSOLUTE |
83203|         | RTL_REGISTRY_OPTIONAL,
83204|         | RegistryPath,
83205|         | &paramTable[0],
83206|         | NULL,
83207|         | NULL
83208|         | ))) {
83209|         Debug(DEBUG_INFO,("Error %08x reading registry
83210|             | key '%S'\n",Status, RegistryPath));
83211|         if(Result->Buffer) {
83212|             RtlZeroMemory(Result->Buffer,Result->MaximumLength);
83213|             Result->Length =
83214|                 | wcslen(Default)*sizeof(WCHAR);
83215|             RtlCopyMemory(Result->Buffer,Default,Result->Length);
83216|             Debug(DEBUG_INFO,("Setting default of
83217|                 | '%S'\n",Result->Buffer));
83218|         } else {
83219|             Debug(DEBUG_INFO,("Error! Out of
83220|                 | memory\n"));
83221|         }
83222|     }

```

```

83214| } else {
83215|
83216|     | Debug(DEBUG_INFO,("Value='%S'\n",Result->Buffer));
83217| }
83218| }
83219|
83220| void Reg_FreeString( PUNICODE_STRING String )
83221| {
83222|     String->Length = String->MaximumLength = 0;
83223|     ExFreePool(String->Buffer);
83224|     return;
83225| }
83226|
83227|
83228| // taken from psm\log.cpp
83229| NTSTATUS LogError( PDEVICE_OBJECT DeviceObject,
83230|                   PIRP Irp,
83231|                   ULONG MessageId,
83232|                   NTSTATUS Error,
83233|                   PVOID DumpData,
83234|                   ULONG DumpDataSize, // in bytes!!!
83235|                   WCHAR *Strings[],
83236|                   ULONG NumStrings )
83237| {
83238|     PIO_ERROR_LOG_PACKET errorLogEntry=NULL;
83239|     PIO_ERROR_LOG_PACKET LogEntry=NULL;
83240|     PWCHAR insertionString=NULL;
83241|     USHORT StrLens=0;
83242|     ULONG i;
83243|     NTSTATUS Status=STATUS_SUCCESS;
83244|     USHORT LogSize=0;
83245|     USHORT SafeDumpSize=0;
83246|     USHORT DumpOffset=0;
83247|
83248|     for(i=0;i<NumStrings;i++) {
83249|
83250|         | StrLens+=wcslen(Strings[i])*sizeof(WCHAR)+sizeof(WCHAR);
83251|         | // add in null character
83252|     }
83253|     // round up to nearest dword length
83254|     SafeDumpSize = ((USHORT)((DumpDataSize+3) /
83255|         | sizeof(ULONG)) * sizeof(ULONG));
83256|
83257|     /*
83258|     typedef struct _IO_ERROR_LOG_PACKET {
83259|         UCHAR MajorFunctionCode;        //0
83260|         UCHAR RetryCount;                1
83261|         USHORT DumpDataSize;             2

```



```

83260| USHORT NumberOfStrings;          4
83261| USHORT StringOffset;              6
83262| USHORT EventCategory;             8
83263| NTSTATUS ErrorCode;               c
83264| ULONG UniqueErrorValue;           10
83265| NTSTATUS FinalStatus;             14
83266| ULONG SequenceNumber;             18
83267| ULONG IoControlCode;              1c
83268| LARGE_INTEGER DeviceOffset;        20
83269| ULONG DumpData[1];                28
83270|                                   2c
83271| }IO_ERROR_LOG_PACKET, *PIO_ERROR_LOG_PACKET;
83272| */
83273|
83274| /*lint -save -e734 -e545 */
83275| DumpOffset =
    | (USHORT)FIELD_OFFSET(IO_ERROR_LOG_PACKET,DumpData);
83276| /*lint -restore */
83277|
83278| LogSize = DumpOffset;    // Have it start
    | here(28) instead of after (30)
83279| LogSize += SafeDumpSize; // number of bytes
    | needed rounded to dword
83280| LogSize += Strlens;      // length of strings
83281|
83282| // can get bigger than a byte (255)
83283| ASSERT(LogSize<=ERROR_LOG_MAXIMUM_SIZE);
83284|
83285| // make sure at least as big as structure (its
    | aligned funny)
83286| if(LogSize<sizeof(IO_ERROR_LOG_PACKET)) {
83287|     LogSize = sizeof(IO_ERROR_LOG_PACKET);
83288| }
83289|
83290| errorLogEntry = (PIO_ERROR_LOG_PACKET)
    | ExAllocatePoolWithTag( PagedPool,LogSize,'TNVE');
83291|
83292| if(errorLogEntry) {
83293|     // mandatory
83294|     errorLogEntry->ErrorCode = MessageId;
83295|
83296|     // optional
83297|     errorLogEntry->StringOffset = NumStrings>0 ?
        | DumpOffset+(USHORT)(SafeDumpSize) : 0;
83298|     errorLogEntry->NumberOfStrings =
        | (USHORT)NumStrings;
83299|     errorLogEntry->DumpDataSize =
        | (USHORT)SafeDumpSize;
83300|     errorLogEntry->SequenceNumber = 0;
83301|     errorLogEntry->RetryCount = 0;

```

```

83302|     errorLogEntry->UniqueErrorValue = 0;
83303|     errorLogEntry->FinalStatus = Error;
83304|     errorLogEntry->EventCategory = 0;
83305|
83306|     if(Irp) {
83307|         PIO_STACK_LOCATION IrpStack =
83308|         | IoGetCurrentIrpStackLocation(Irp);
83309|         errorLogEntry->MajorFunctionCode =
83310|         | IrpStack->MajorFunction;
83311|         if( (IrpStack->MajorFunction ==
83312|         | IRP_MJ_DEVICE_CONTROL) ||
83313|         | (IrpStack->MajorFunction ==
83314|         | IRP_MJ_INTERNAL_DEVICE_CONTROL) ||
83315|         | (IrpStack->MajorFunction ==
83316|         | IRP_MJ_SCSI) ) {
83317|             errorLogEntry->IoControlCode =
83318|             | IrpStack->Parameters.DeviceIoControl.IoControlCode;
83319|             errorLogEntry->DeviceOffset.QuadPart =
83320|             | 0;
83321|         } else {
83322|             errorLogEntry->DeviceOffset =
83323|             | IrpStack->Parameters.Read.ByteOffset;
83324|             errorLogEntry->IoControlCode = 0;
83325|         }
83326|     } else {
83327|         errorLogEntry->MajorFunctionCode = 0 ;
83328|         errorLogEntry->IoControlCode = 0;
83329|         errorLogEntry->DeviceOffset.QuadPart = 0;
83330|     }
83331|
83332|     if(DumpDataSize>0) {
83333|         | RtlZeroMemory(errorLogEntry->DumpData, SafeDumpSize);
83334|         | RtlMoveMemory(errorLogEntry->DumpData, DumpData, DumpDataS
83335|         | ize);
83336|     }
83337|
83338|     insertionString =
83339|     | (PWSTR)((PCHAR)(errorLogEntry) +
83340|     | errorLogEntry->StringOffset);
83341|
83342|     for(i=0;i<NumStrings;i++) {
83343|         RtlCopyMemory(insertionString, Strings[i],
83344|         | wcslen(Strings[i])*sizeof(WCHAR)+sizeof(WCHAR));
83345|         insertionString+=wcslen(Strings[i])+1; //
83346|         | in WCHARs

```

```

83337|     }
83338|
83339|     // we do this so we can check we dont overwrite
      | our buffers.
83340|     LogEntry = (PIO_ERROR_LOG_PACKET)
      | IoAllocateErrorLogEntry( DeviceObject, (UCHAR)LogSize);
83341|     if(LogEntry) {
83342|
      | RtlMoveMemory(LogEntry,errorLogEntry,LogSize);
83343|         IoWriteErrorLogEntry (LogEntry);
83344|     } else {
83345|         Status = STATUS_INSUFFICIENT_RESOURCES;
83346|     }
83347|
83348|     ExFreePool(errorLogEntry);
83349|     Status=STATUS_SUCCESS;
83350| } else {
83351|     Status = STATUS_INSUFFICIENT_RESOURCES;
83352| }
83353| return Status;
83354| }
83355|
83356|
83357| #ifdef NOT_USED
83358| NTSTATUS GetDriveLetterFromNTName( WCHAR *NTName, WCHAR
      | *DriveString, ULONG BufferSize )
83359| {
83360|     WCHAR      DriveName[20] = L"\\DosDevices\\C:";
83361|     UNICODE_STRING UniName={0};
83362|     OBJECT_ATTRIBUTES ObjAttr={0};
83363|     HANDLE      SymHandle=NULL;
83364|     WCHAR      SymSpace[256]={0};
83365|     NTSTATUS      Status=STATUS_UNSUCCESSFUL;
83366|     ULONG      Ret=0;
83367|     UCHAR      DriveLetter=0;
83368|     NTSTATUS RetStatus = STATUS_NOT_FOUND;
83369|
83370|     PAGED_CODE();
83371|
83372|     for ( DriveLetter=2;DriveLetter<26;DriveLetter++ )
      | {
83373|
83374|         DriveName[12] = DriveLetter+65;
83375|
83376|         RtlInitUnicodeString( &UniName, DriveName );
83377|
83378|         ObjAttr.Length          =
      | sizeof(ObjAttr);
83379|         ObjAttr.RootDirectory    = NULL;
83380|         ObjAttr.Attributes      =

```

```

    | OBJ_CASE_INSENSITIVE;
83381|     ObjAttr.ObjectName          = &UniName;
83382|     ObjAttr.SecurityDescriptor  = NULL;
83383|     ObjAttr.SecurityQualityOfService = NULL;
83384|
83385|     Status = ZwOpenSymbolicLinkObject( &SymHandle,
    | STANDARD_RIGHTS_READ, &ObjAttr );
83386|     if ( NT_SUCCESS(Status) ) {
83387|         UniName.Length = 0;
83388|         UniName.MaximumLength = 256;
83389|         UniName.Buffer = SymSpace;
83390|         Status =
    | ZwQuerySymbolicLinkObject(SymHandle,&UniName,&Ret);
83391|         if ( NT_SUCCESS(Status) ) {
83392|             Debug(DEBUG_DCPSM,("Symlink '%S' ==
    | '%wZ\\n",DriveName,&UniName));
83393|
83394|             if (
    | _wcsnicmp(UniName.Buffer,NTName,wcslen(NTName))==0 ) {
83395|                 // \\DosDevices\C:
83396|                 // 0123456789-12
83397|                 DriveString[0] = DriveLetter+65;
83398|                 DriveString[1] = 0;
83399|                 ZwClose(SymHandle);
83400|                 RetStatus= STATUS_SUCCESS;
83401|                 break;
83402|             }
83403|         } else {
83404|             Debug(DEBUG_DCPSM,("Error
    | %08x\\n",Status));
83405|         }
83406|
83407|         ZwClose(SymHandle);
83408|     } else {
83409|         //Debug(DEBUG_DCPSM,("Error
    | %08x\\n",Status));
83410|     }
83411| } // for
83412|
83413| return RetStatus;
83414| }
83415|
83416| NTSTATUS Sblo_OpenVolumeHandle (
83417|     const WCHAR    *VolumeName,
83418|     HANDLE         &VolumeHandle,
83419|     PFILE_OBJECT   &VolumeFileObject,
83420|     ULONG          DesiredAccess )
83421| {
83422|     UNICODE_STRING  UniName={0};
83423|     OBJECT_ATTRIBUTES ObjectAttributes={0};

```

```

83424| IO_STATUS_BLOCK    IoStatus = {0};
83425| HANDLE              FileHandle = 0;
83426| NTSTATUS             Status = 0;
83427|
83428| VolumeHandle = INVALID_HANDLE_VALUE;
83429| VolumeFileObject = NULL;
83430|
83431| RtlInitUnicodeString(&UniName,VolumeName);
83432|
83433| InitializeObjectAttributes (
83434|     &ObjectAttributes,
83435|     &UniName,
83436|     OBJ_CASE_INSENSITIVE,
83437|     NULL,
83438|     NULL );
83439|
83440| Status = ZwCreateFile(
83441|     &VolumeHandle,
83442|     DesiredAccess,
83443|     &ObjectAttributes,
83444|     &IoStatus,
83445|     NULL,                                //
    | alloc size
83446|     FILE_ATTRIBUTE_NORMAL,
83447|     FILE_SHARE_WRITE | FILE_SHARE_READ,
83448|     FILE_OPEN,
83449|     0,                                    //
    | create options
83450|     NULL,                                //
    | eabuffer
83451|     0 );                                //
    | ealength
83452|
83453| if(NT_SUCCESS(Status)) {
83454|     Status = ObReferenceObjectByHandle(
83455|         VolumeHandle,
83456|         GENERIC_WRITE,
83457|         NULL,
83458|         (KPROCESSOR_MODE)KernelMode,
83459|         (PVOID *)&VolumeFileObject,
83460|         NULL );
83461|
83462|     if(!NT_SUCCESS(Status)) {
83463|         Debug(DEBUG_DEVSUP,("Sblo_OpenVolumeHandle:
    | Error %08x in
    | ObReferenceObjectByHandle('%S')\n",Status,VolumeName));
83464|         ZwClose (VolumeHandle);
83465|         VolumeHandle = INVALID_HANDLE_VALUE;
83466|         VolumeFileObject = NULL;
83467|     }

```

```

83468|     } else {
83469|         Debug(DEBUG_DEVSUP,("Sblo_OpenVolumeHandle:
| Error %08x in
| ZwCreateFile("%S")\n",Status,VolumeName));
83470|         VolumeHandle = INVALID_HANDLE_VALUE;
83471|     }
83472|
83473|     return Status;
83474| }
83475|
83476|
83477| NTSTATUS Sblo_CloseVolumeHandle (
83478|     HANDLE      &VolumeHandle,
83479|     PFILE_OBJECT &VolumeFileObject )
83480| {
83481|     NTSTATUS Status = STATUS_SUCCESS;
83482|     if ( VolumeFileObject==NULL ||
| !IsValidHandle(VolumeHandle) ) {
83483|         ASSERT ( VolumeFileObject != NULL );
83484|         ASSERT ( IsValidHandle(VolumeHandle) );
83485|         Status = STATUS_INVALID_PARAMETER;
83486|         Debug(DEBUG_DEVSUP,("Sblo_CloseVolumeHandle:
| Invalid parameter -- VolumeHandle=%08x,
| VolumeFileObject=%08x\n",VolumeHandle,VolumeFileObject))
| ;
83487|     } else {
83488|         ObDereferenceObject (VolumeFileObject);
83489|         ZwClose (VolumeHandle);
83490|
83491|         VolumeFileObject = NULL;
83492|         VolumeHandle = INVALID_HANDLE_VALUE;
83493|     }
83494|
83495|     return Status;
83496| }
83497|
83498|
83499| NTSTATUS Sblo_DeviceIoControl( PDEVICE_OBJECT
| DeviceObject,
83500|     ULONG IoctlCode,
83501|     char *InBuffer,
83502|     ULONG InBufferSize,
83503|     char *OutBuffer,
83504|     ULONG OutBufferSize,
83505|     ULONG *BytesReturned
83506|     )
83507| {
83508|     PIRP Irp=NULL;
83509|     IO_STATUS_BLOCK IoSb={0};
83510|     KEVENT Event={0};

```

```

83511| NTSTATUS Status=STATUS_UNSUCCESSFUL;
83512|
83513| *BytesReturned = 0;
83514|
83515| // Set the event object to the unsigaled state.
83516| // It will be used to signal request completion.
83517|
83518| KeInitializeEvent(&Event, NotificationEvent,
| FALSE);
83519|
83520| Irp = IoBuildDeviceIoControlRequest( IoctlCode,
83521| DeviceObject,
83522| InBuffer,
83523| InBufferSize,
83524| OutBuffer,
83525| OutBufferSize,
83526| FALSE,
83527| &Event,
83528| &IoCb);
83529|
83530|
83531| if(Irp) {
83532|
83533|     __try {
83534|         Status = IoCallDriver(DeviceObject, Irp);
83535|
83536|         if (Status == STATUS_PENDING) {
83537|
83538|             ASSERT(KeGetCurrentIrql() <
| DISPATCH_LEVEL);
83539|             pmWaitForSingleObject(&Event,NULL);
83540|
83541|             Status = IoCb.Status;
83542|         }
83543|     } __except(EXCEPTION_EXECUTE_HANDLER) {
83544|         Status = GetExceptionCode();
83545|
83546|         | Debug(DEBUG_DEVSUP,("Sblo_DeviceIoControl:Error!
| Exception %08x sending scsi command\n",Status));
83547|     } else {
83548|         Debug(DEBUG_INFO,("Sblo_DeviceIoControl: Error
| no Irp available\n"));
83549|         Status = IoCb.Status;
83550|     }
83551|
83552|     if (!NT_SUCCESS(Status)) {
83553|         *BytesReturned = IoCb.Information;
83554|     }
83555|

```

```

83556|
83557|     return Status;
83558| }
83559| #endif
83560|
83561| /*--- end of file support.cpp ---*/
83562|
83563|
83564|
83565| File Listing: support.h
83566|
83567| extern "C" {
83568|     void Reg_GetULONGKey (
83569|         IN PWCHAR RegistryPath,
83570|         IN PWCHAR Key,
83571|         IN ULONG Default,
83572|         OUT PULONG Result
83573|     );
83574|
83575|     NTSTATUS LogError( PDEVICE_OBJECT DeviceObject,
83576|         PIRP Irp,
83577|         ULONG MessageId,
83578|         NTSTATUS Error,
83579|         PVOID DumpData,
83580|         ULONG DumpDataSize, // in bytes!!!
83581|         WCHAR *Strings[],
83582|         ULONG NumStrings
83583|     );
83584|     NTSTATUS GetDriveLetterFromNTName( WCHAR *NTName,
83585|         | WCHAR *DriveString, ULONG BufferSize );
83586|     void Reg_GetStringKey (
83587|         IN PWCHAR RegistryPath,
83588|         IN PWCHAR Key,
83589|         IN PWCHAR Default,
83590|         OUT PUNICODE_STRING Result
83591|     );
83592|     void Reg_FreeString( PUNICODE_STRING String );
83593|
83594| }
83595|
83596| /*--- end of file support.h ---*/
83597|
83598|

```